Increasing Predictive Accuracy by Prefetching Multiple Program and User Specific Files

Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt
Computer Science Department
University of California, Santa Cruz
1156 High Street, Santa Cruz, CA 95064
{yeh,darrell,sbrandt}@cs.ucsc.edu

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 and User-based Last n Successors (PULnS) *file prediction model that identifies relationships between files through the names of the programs and the users accessing them. Our simulation results show that, in the worst case, PULnS makes at least 20% fewer incorrect predictions and roughly the same number of correct predictions as the last-successor model.*

1 Introduction

As disks operate significantly more slowly 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. In addition to saving disk traffic [16], caching is also a technique commonly used to improve the performance of distributed file systems [20, 15, 6].

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, which can reduce the program stall-time. On the other hand it will consume additional cache space and disk bandwidth, which can bring down the system performance if it is not done wisely. Thus it is important 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 [10, 12, 4, 5, 11, 17], as have hints or help from programs and compilers [18, 3].

Files are accessed by programs, and programs are executed on behalf of users. This suggests that consecutive accesses to different files should not occur without reasons. Our earlier works [22, 23] have shown 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 to different files can be more accurately predicted given knowledge about which programs and users are accessing them.

This paper presents the Program-based and User-based Last n Successors (PULnS) file prediction model, a refinement of our earlier work, Program-based and User-based Last Successor (PULS) [23], based on the observation that program-based and user-based successors could still change in some cases. The n in PULnS indicates the maximum number of files PULnS can predict per prediction.

We compare PULnS with Last-Successor (LS) for different values of "n" in PULnS. Our experiments show that, compared with LS, PULnS makes more correct predictions and fewer incorrect predictions, which should improve the overall performance in a real system. Our results also show that PUL2S ($n = 2$) can reduce as much as 48% of incorrect predictions made by LS with the cost of predicting 1.36 files per event on average. As a result, PUL2S provides the best cost-effective performance in terms of the number of files

Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)
0-7695-1626-2/02 $17.00 © 2002 IEEE

predicted each time and the predictive accuracy improved.

We also examine the cache hit ratios of Least Recently Used (LRU) with no file prediction, and LRU with PULnS. We observe that PULnS always increases the cache hit ratio, and in the best case, LRU and PUL1S ($n = 1$) together perform almost as well as a cache up to 80 times larger using LRU alone.

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 and user information like PULnS 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 the 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.* [21], 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 PULnS is that PULnS makes the predicting decision for each individual file, so it can adapt to different patterns of file access more rapidly.

Vitter *et al.* adopt the technique of data compression to predict the next required page [4, 21]. 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 [18]. 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]. Mowry *et al.* use a modified compiler to provide future access patterns for out-of-core applications [13].

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 [17]. Kimbrel *et al.* examine four related algorithms to find out when a prefetching algorithm should act aggressively or conservatively [7].

3 LS and PULnS Models

We start with a brief description of the LS model, followed by the discussion of the reasons why PULnS is a better model than LS, and finally we explain how to build PULnS.

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 PULnS (Program-based and User-based Last n Successors)

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. However, probability can only tell us what patterns of file accesses are and how frequently they occur, but not why these patterns exist. One disadvantage of using the probability 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. Consequently, file prediction algorithms should not adopt the system-wide history of file accesses to make predictions. Even the system-wide history of file

accesses is changeable when we repeat executing the same set of programs, however, what could remain unchanged is the order of files accessed by the individual programs, particularly when each program is executed by the same user, which can be useful in predicting what files will be used next for individual programs. In other words, file reference patterns can describe what has happened more precisely if they are observed for each individual program together with the user executing that program. As a result, better knowledge about past access patterns leads to better predictions of future access patterns.

PULnS incorporates the knowledge about running programs and corresponding users to generate a better successor estimate. More precisely, PULnS records and predicts successors specific to programs and users for each file that is accessed. The n in PULnS represents the number of the most recent distinct program-specific and user-specific successors that PULnS could predict each time. For example, PUL1S ($n = 1$) means that only the most recent successor of this type is predicted after each file access. In other words, PUL1S can be viewed as the PULS model discussed in our earlier work [23]. PUL2S ($n = 2$) predicts the most recent two successors observed this way when there are more than one existing for a given file. However, it still predicts only one successor if the program-specific and user-specific successor for a given file has never changed.

We will use PUL1S as an example to explain how PULnS works. 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 the user U_1 executing the program P_1, while the pattern AC is caused by U_1 running another program, P_2. In the meanwhile, pattern AD comes from the case when the user U_2 executes P_1, and pattern AE will be observed when the program P_2 is executed on behalf of user U_3. In other words, what is really behind these four patterns of consecutive file accesses is the execution of two different applications, P_1 and P_2, executed on behalf of three different users, U_1, U_2 and U_3. After we collect this 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 executed on behalf of a given user accesses multiple different files after each access of a particular file, then the corresponding successors will change.

Similarly, PUL2S ($n = 2$) predicts the most recent two distinct program-specific and user-specific successors if the program-specific and user-specific successor ever changed. For example in Figure 2, if program P_1 executed by U_1 accesses a different file, F, other than B after an access to A. PUL1S will predict F from now on, while PUL2S will predict both F and B since they are the most recent two distinct successors that file A keeps for program P_1 and user U_1.

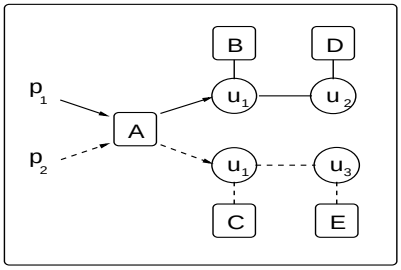

Figure 1. PUL1S model

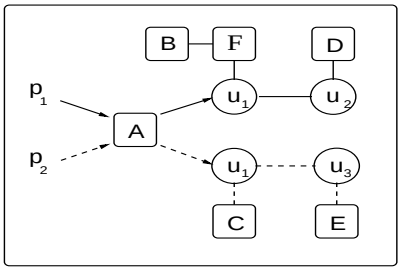

Figure 2. PUL2S model

Table 1. Metadata of Figure 1 kept under the PUL1S 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	$\langle P_2, NIL\rangle$

There are two 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 P has most recently accessed. When P accesses the next file (say Y) after X, we update the metadata of X with $\langle P, UY\rangle$, and the next time that P accesses X on behalf of U, PUL1S can predict that the next file accessed will be Y.

In the example of Figure 1, 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, PUL1S 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 1 is shown in Table 1.

In the case of PUL2S ($n = 2$), we keep the most recent two distinct program-specific and user-specific successors for each file. So, for the file A in Figure 2, the corresponding metadata $\langle P_1, U_1B\rangle$ now becomes $\langle P_1, U_1FB\rangle$.

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 and user-specific successors to the file, so that we know which file (or files) to predict when the same program executed by a particular 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 majority of files are accessed by five or fewer programs and thus metadata storage is not a problem.

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 subprograms at different times is considered by PULnS to be a different program from the subprograms, each of which is also treated independently. The second is that ideally both "*program name*" and "*file name*" should include the entire pathname of the files so files with the same name in different directories can be handled appropriately. In practice, we may replace the full-path file name with its inode (index node) number or we can apply a simple hash function to the full-path file name so we do not have to store the full-path file name for each file.

4 Experimental Results

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

4.1 Simulation Trace

The key requirement of the file trace we need is the information of corresponding programs and users initiating events of file access recorded in the trace. User information is available in some traces we studied. However, the program information was not recorded in all the file traces we have access to, except the *DFSTrace* from the *Coda* project [8, 14]. Thus we selected 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 from September 1992 to February 1993 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. Because DFSTrace does not record events of *READ* or *WRITE* in most cases. Therefore, instead of tracking every *READ* or *WRITE* event, we track only the *FORK*, *EXECVE* and *OPEN* events in our simulation.

As mentioned above, PULnS needs to be able to determine the name of a program and its user in order to generate the 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. Similarly, we also have to exclude *OPEN* events initiated by any *process ID* (PID) which started before the beginning of our trace due to the unknown program name for that PID. 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.

One may claim that the *DFSTrace* does 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, most file accesses are sequential [1], 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 are still adequate to evaluate our algorithm.

4.2 Methodology of Performance Evaluation

We are interested in how different values of "n" in PULnS could affect the performance and the related costs when compared with LS. We used the filtered trace data to evaluate LS, PUL1S, PUL2S, and PUL3S respectively.

Both LS and PUL1S predict one file at a time. We score LS and PUL1S by adding one for each correct prediction and zero for each incorrect prediction to their total scores. We normalize the final scores of PUL1S and LS to the number of predictions, not to 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. Since our simulation trace is very long

(six months), it turns out that the effect of this compulsory error is negligible and does not affect the comparison of predictive accuracy in our experiments. We also score PUL2S and PUL3S the same way we score LS and PUL1S. A more detailed discussion of PUL2S and PUL3S will follow later.

4.3 Comparison of Predictive Accuracy

Figure 3 displays the comparison of predictive accuracy between LS and PULnS. It is clear that PUL1S delivers a higher predictive accuracy than LS in all machines. Both PUL2S and PUL3S also perform better than LS. It is worth pointing out that PUL2S outperforms PUL1S more than PUL3S outperforms PUL2S, which indicates that keeping two program-based and user-based last successors for each file can be a cost-effective way to increase the predictive accuracy.

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 not making any 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 not making 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 PULnS in three categories. The first category is the percentage of total predictions (including correct and incorrect predictions) made by PULnS 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 PULnS as compared with LS. This number should be as high as possible. The last category is the percentage of incorrect predictions made by PULnS as compared with LS. Ideally this percentage should be less than 100%, indicating that PULnS model makes fewer incorrect predictions than LS.

4.4 Performance by Category

Figure 4 displays the performance in the category of total prediction. As a reminder, members of PULnS family mainly differ in the maximum number, n, of files they could predict each time. Thus, the number of cases where a prediction was made is the same in the three members of PULnS family we examined. For simplicity, we only show the data from PUL1S. It indicates that the percentage

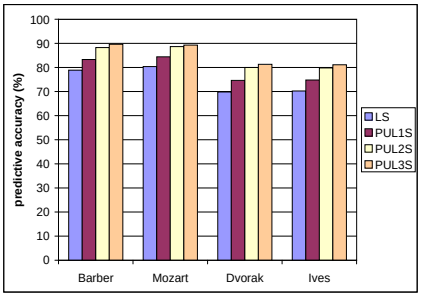

Figure 3. Predictive accuracy of LS and PULnS

of events where a prediction was made by PUL1S is only about five to seven percent less than that of LS. This is close enough to consider PULnS to be a practical prediction algorithm in terms of the number of predictions it makes. The percentage of correct prediction is shown in Figure 5. Figure 5 demonstrates that PUL1S can do roughly as well as LS in correctly predicting files, while PUL2S and PUL3S can do better than LS.

Figure 6 shows the percentage of incorrect prediction. To get a closer look at the relative performance in terms of reducing incorrect predictions, data in Figure 6 is normalized to that of LS and displayed in Figure 7 (that is why the percentage of LS is 100% in Figure 7). Figure 7 clearly demonstrates that PULnS family can make significantly fewer incorrect predictions than LS. PUL1S reduces about 21% (Dvorak) to 27% (Barber) of incorrect prediction compared with LS. PUL2S can do approximately 37% to 48% less (in Ives and Barber), while PUL3S can do about 42% to 55% less (in Dvorak and Barber). As we discussed before, incorrect predictions come with a cost, and avoiding this cost directly translates into better system performance.

As mentioned earlier, PUL2S and PUL3S could predict two or three file per prediction respectively. However, the program-based and user-based last successors can remain unchanged in some cases, which could lower the average number of files predicted per event (*i.e.* per file access) for PUL2S and PUL3S. Take Barber for example, our simulation shows that the average numbers of files predicted per event for PUL1S, PUL2S, and PUL3S are 0.92, 1.36, and 1.64 respectively. Obviously PUL2S provides a better cost-effective performance than PUL3S when we consider the number of files predicted each time and the performance improvement comes with it.

Because we do not count the cases where no prediction was made when calculating predictive accuracy, so the number of events in the trace is larger than the number of cases where a prediction was made. Because PUL1S predicts one

file at a time whenever a prediction was made, so the number of files predicted per event is smaller than one (0.92) for PUL1S. Similarly, the numbers for PUL2S and PUL3S are smaller than two and three respectively.

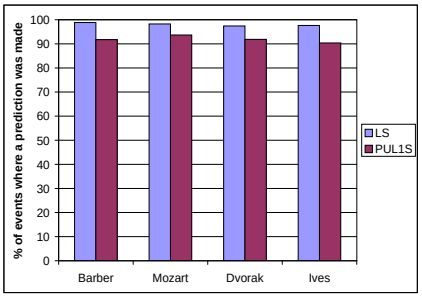

Figure 4. Total prediction made by LS and PUL1S

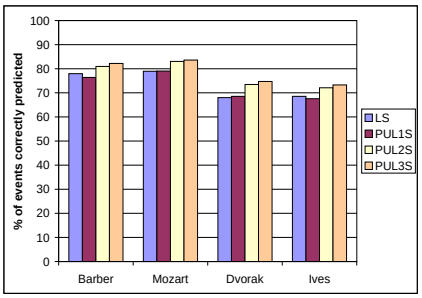

Figure 5. Correct prediction made by LS and PULnS

Figure 5 and Figure 6 indicate that file prediction can be more accurate if we take advantage of the user and program information. To understand the underlying reason for the performance improvement in PULnS, we collect the number of distinct program-based and user-based successors for each file that has at least one successor of this type in our simulation. The result is listed in Table 2. We will use the percentage collected from Barber for discussion.

The first row in Table 2 shows that 42.33% of the files have only one program-based and user-based successor observed through the entire simulation. In other words, their program-based and user-based successors never change. For those cases, if we know the file access is initiated by which program and the user executing that program, PUL1S can always correctly predict the next file needed by that program. The second row shows that 13.52% of files have two distinct program-based and user-based successors. For those files, PUL1S will make incorrect prediction in some

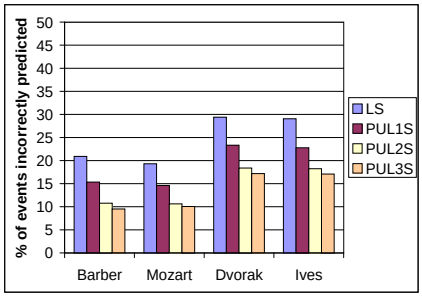

Figure 6. Incorrect prediction made by LS and PULnS

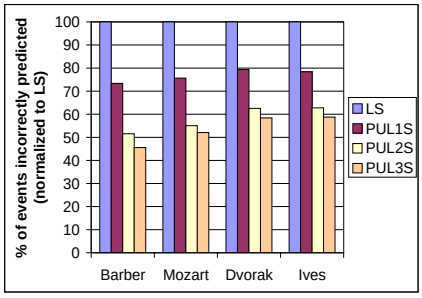

Figure 7. Incorrect prediction made by LS and PULnS (normalized to LS)

cases, however, PUL2S can assure the success of the prediction it makes. The last row shows that about 18.91% of the files have observed three or more program-based and user-based successors.

Table 2 uncovers an important fact – most files have a limited number of program-specific and user-specific successors, which is the underlying reason for the performance improvement achieved by PULnS. As a result, even in the worst case, PUL3S can guarantee the success of at least 70% to 80% of the predictions it made in our simulation.

Table 2. The number of program-based and user-based successors observed for each file

number	Barber	Mozart	Dvorak	Ives
1	42.33%	16.70%	21.83%	20.43%
2	13.52%	10.51%	14.48%	12.03%
3	25.24%	41.53%	35.38%	37.73%
3+	18.91%	31.26%	28.31%	29.81%

One last note about the percentage of correct prediction in Figure 5. The common measurement of correct predic-

tion is very simple. If the predicted file is the next file needed by the system, regardless which program initiates the access to that file, then it is considered as a correct prediction. Otherwise it is viewed as an incorrect prediction. In our simulation, we use this common measurement to obtain the percentage of correct prediction as seen in Figure 5. However, this approach does not fully reflect the real performance of PULnS. In general, a system could have multiple programs in execution concurrently. A program-specific and user-specific prediction made for one program is not likely to be the same file predicted for other programs, much less when they are executed by different users. Besides, programs are often interrupted for the sake of cache miss or fair scheduling during their execution. Consequently the file predicted for the current program may not satisfy the file access coming up next initiated by other running programs in the system. In other words, the percentage of correct prediction in Figure 5 (or the predictive accuracy in Figure 3) shows the performance of PULnS when the cache memory in the system can only hold one predicted file. Since modern computer systems have cache memory large enough to hold more than one predicted file, PULnS should perform better than what Figure 5 demonstrates in a real system. We can obtain a more realistic performance of PULnS by measuring the cache hit ratio when PULnS is applied. We will discuss this in the next section.

4.5 Cache Performance

As explained in the previous section, we want to know how PULnS performs in terms of cache hit ratio besides its percentage of correct prediction observed by the common measurement. 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 [19]. The second is that in the case of large files, sequential read is the most common activity. As we stated earlier, 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 PUL1S. The reason we chose PUL1S is that it predicts one file per prediction, so we can get the conservative estimation of the real performance of PULnS. LRU has been widely employed [24], so it is an appropriate candidate used to evaluate the effectiveness of prediction algorithms in terms of cache hit ratio. Figure 8 shows that when using PUL1S prediction, the cache always performs better than when using LRU alone, regardless of cache size, and in the best case (Barber) it performs almost as well as a cache up to 80 times larger using LRU alone (cache hit ratio 89.37% for cache size 25 when PUL1S applied, versus 89.81% for cache size 2000 using LRU alone).

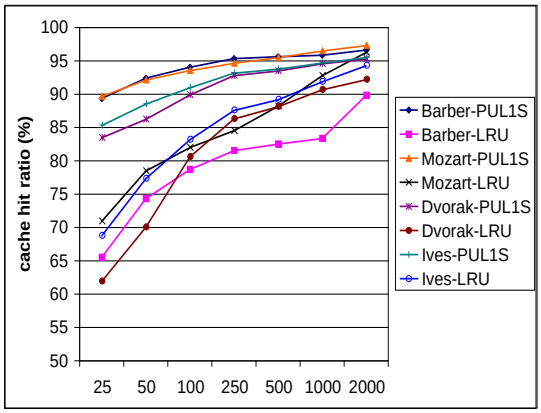

Figure 8. Cache hit ratio of LRU (lablled LRU) and LRU with PUL1S (labelled PUL1S)

5 Future Work

The DFSTrace is almost 10 years old. We chose it because it contains the program and user information, which is absolutely necessary to the PULnS model. In the future, we would like to collect our own traces that PULnS can use, and examine how PULnS performs under more recent traces. Ultimately, we will build the PULnS into the filesystem and evaluate its performance in a real system. There are still some alternatives may improve the performance of PULnS and are worthy of further exploration. For example, files existing temporarily (such as those in */tmp* directory) usually will not get the same name next time they are created again. If so, then they can never be predicted correctly by PULnS, and there is no need to store their information.

6 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.

File accesses are driven by the programs and the users using them, not by previous access patterns. By tracking programs and users initiating file accesses, we successfully avoid many incorrect predictions. We show that PULnS can reduce a significant percentage of incorrect prediction made by LS. Compared with LS, about 37% to 48% of incorrect predictions can be reduced in PUL2S as seen in Figure 7. Consequently, the overall performance penalty in a system caused by incorrect predictions can be significantly reduced. With the cost of predicting 1.36 files per event on average, we can conclude that PUL2S provides a better cost-effective performance than PUL3S. We also compare the cache hit ratios of LRU with and without PUL1S. The results show that with PUL1S, LRU can deliver a much higher cache hit ratio.

7 Acknowledgments

We are 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] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. In *ACM Transactions of Computer Systems*, pages 51–81, 1988.

[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] 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.

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

[15] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. In *ACM Transactions of Computer Systems*, pages 228–239, 1993.

[16] J. Ousterhout, H. D. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 15–24, 1985.

[17] 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.

[18] 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.

[19] 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.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of USENIX Annual Technical Conference*, pages 119–130, 1985.

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

[22] 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.

[23] T. Yeh, D. D. E. Long, and S. Brandt. Using Program and User Information to Improve File Prediction Performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 111–119, 2001.

[24] 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.