Title

Performing File Prediction with a Program-Based Successor Model

Authors

Yeh, Tsozen
Long, Darrell DE
Brandt, Scott A

Publication Date

2001

DOI

10.1109/mascot.2001.948869

Peer reviewed

Performing File Prediction with a Program-Based Successor Model

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

Recent increases in CPU performance have surpassed those in hard drives. As a result, disk operations have become more expensive in terms of the number of CPU cycles spent waiting for them to complete. File prediction can mitigate this problem by prefetching files into cache before they are accessed. Identifying relationships between individual files plays a key role in successfully performing file prefetching. It is well-known that previous patterns of file references can be used to predict future references. Nevertheless, knowledge about the programs producing the relationships between individual files has rarely been investigated. We present a *Program-Based Successor* (PBS) model that identifies relationships between files through the names of the programs accessing them. We develop a *Program-based Last Successor* (PLS) model derived from PBS to do file prediction. Our simulation results show that PLS makes 21% fewer incorrect predictions and roughly the same number of correct predictions as the *Last-Successor* (LS) model. We also examine the cache hit ratio achieved by applying PLS to the *Least Recently Used* (LRU) caching algorithm and show that a cache using PLS and LRU together can perform better than a cache up to 40 times larger using LRU alone. Finally, we argue that because program-based successors are more likely to be used soon, incorrectly prefetched program-based successors are more likely to be used and thus less incorrect than incorrectly prefetched files from non-program-based models.

Keywords – file prediction, prefetching

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

1 Introduction

Running programs stall if the data they need is not in memory. As CPU speeds increase, disk I/O becomes more expensive in terms of the CPU cycles spent waiting for the data to be read from disk. File prefetching is a technique that mitigates this speed difference, originating from the mechanical operation of disk and the electronic operation of CPU [17], by preloading files into memory before they are needed. The success of file prefetching ultimately depends on the accuracy of the file prediction algorithm – how accurately an operating system can predict which files to load into memory. Probability and history of file access have been widely used to perform file prediction [4, 5, 9–11, 14], as have hints or help from programs and compilers [3, 12, 15].

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

We present a Program-Based Successor (PBS) model which identifies relationships between individual files through the names of the programs accessing them. We contend that relationships between files are caused by programs. The execution of a program decides both what files it needs, and the accessing order among them. In other words, probability and repeated history of file accesses do not occur for no reason. Most likely they exist as the result of repeated execution of the same program or repeated execution of part of a program. Programs access more or less the same files in roughly the same order every time they execute, so consecutive accesses of different

files can be more accurately predicted given knowledge about which programs are accessing them. By knowing which program is initiating a file access, PBS can provide a more accurate prediction of the next file needed. In addition to prefetching, PBS also has applications in battery conservation in mobile computers and file hoarding in weak-connected or disconnected network environments.

We also present a new file prediction algorithm, Program-based Last Successor (PLS) based on the PBS model. PLS uses knowledge about which programs are accessing the files to determine program-specific last-successor predictions for each file. Our results demonstrate that that PLS generates more accurate file predictions than the other file prediction algorithms examined. In particular, PLS reduces 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 with Last-Successor (LS) and *Finite Multi-Order Context* (FMOC) [9]. Generally speaking, LS has a high predictive accuracy – our simulation results show that LS can correctly predict the next file to be accessed about 80% of the time in some cases. FMOC outperformed LS in a one-month trace in Kroeger's study [9] but performs slightly worse than LS in our simulations. Our experiments demonstrate that with traces covering as long as 13 months PLS makes up to 21.48% fewer incorrect predictions than LS, giving PLS the highest predictive accuracy among all three models in our comparison. We also examine the cache hit ratios of Least Recently Used (LRU) with no file prediction, and LRU with PLS. We observe that PLS always increases the cache hit ratio and in the best case, LRU and PLS together have a better cache hit ratio than a cache 40 times larger using LRU alone.

2 Related Work

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 decision. Their simulations show that different combinations of window and threshold values will largely affect the performance.

Lei and Duchamp use pattern trees to record past execution activities of each program [11]. 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.

Vitter, Curewite, and Krishnan adopt the technique of data compression to predict next required page [4, 18]. Their observation is that data compressors assign a smaller code to the next character with a higher predicted probability. Consequently a good data compressing algorithm should also be good at predicting next page more accurately.

Kroeger and Long predict next file based on probability of files in contexts of FMOC [9]. They later improved FMOC to *Partitioned Context Modeling* (PCM) which limits the total number of nodes in each partition to lower the space requirement in FMOC. Their research also adopts the idea of data compression like Vitter *et al.* [18], but they apply it to predicting next file instead of next page.

Patterson *et al.* develop *TIP* to do prediction using hints provided from modified compilers [15]. 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 [12]. Kotz and Ellis define representative parallel file access patterns in parallel disk systems [8]. 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 [14]. Kimbrel *et al.* examine four related algorithms to find out when a prefetching algorithm should act aggressively or conservatively [6].

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

3 Program-Based Successor (PBS) Model

We provide details about the PBS model in this section. We start with a discussion of why PBS is a useful model,

followed by a discussion of how to implement this model and how to apply it to file prediction.

3.1 PBS

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.

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. In the above example, executing P_1 and P_2 concurrently may result in many different file access patterns. However what remains unchanged is the order of files accessed by the individual programs, P_1 or P_2. This validates our observation that probability and patterns of file access occur for reasons. 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.

We present a Program-Based Successor (PBS) model which keeps track of previous executions of programs and the sequence of files they have accessed. Consequently, file prediction can be carried out in a more precise way. Each file in the PBS model has a record of every program that has accessed it, and the files previously accessed by that program after this one. For a given program, PBS can predict the sequence of files it may access from the records kept with files. Figure 1 shows a simple example of the PBS model. The left and right parts represent the beginning and the end of a program's execution respectively. The middle portion describes files and the order among them accessed by programs. In this figure program P_1 accesses file A, then B. P_2 accesses A and C in that order. P_3 accesses either the sequence of files D, E, then H, or D, F, E, then G.

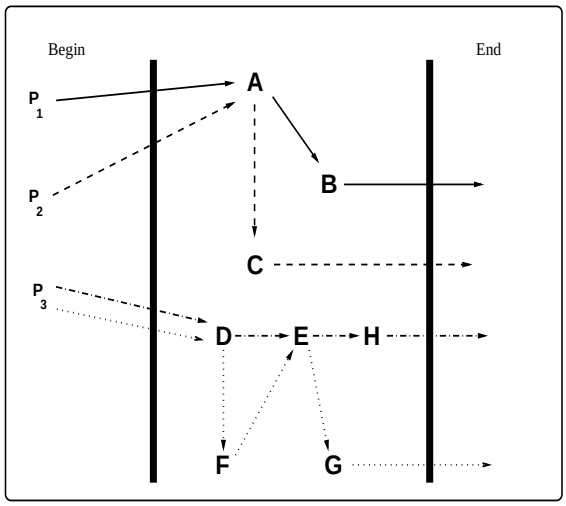

Figure 1: A simple example of the Program-Based Successor model

3.2 Building PBS

The relationship between files and programs accessing them can be many-to-many. Therefore a model must store metadata efficiently to save space and to react promptly. For each file PBS stores pairs of ⟨*program name*, *successor-list*⟩ in the metadata of the file. The *program name* in each pair represents a program which accessed that file before. The *successor-list* is a list of files that the program accessed immediately after previous accesses to the corresponding file. PBS stores information about files needed by different programs efficiently. It does not keep the entire model in memory like some other file prediction models. In addition to the data structure used by running programs to update ⟨*program name*, *successor-list*⟩ of files they access, only the metadata of files that programs are currently accessing need to stay in memory to make predictions. Once a program exits, the metadata used for PBS no longer needs to exist. The metadata of the files in Figure 1 is displayed in Table 1. Details of building and updating metadata of files will be discussed in the next section.

Programs are executed as processes, so we can 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, PBS updates the metadata of the X with ⟨P, Y⟩. If the access to X by P

is ever followed by access to different files, for example Z, other than Y, PBS adds the name of file Z to the metadata of X. So the metatdata now becomes ⟨P, Y, Z⟩.

The metadata within each file, ⟨P, Y⟩ in the above example, will remain with the file until the file is deleted from the file system. The extra metadata each running program keeps for the file it has most recently accessed only exists during the execution of the program. Once a program terminates this metadata is no longer needed and may be discarded.

Table 1: Metadata of Figure 1 kept under PBS model

file	⟨*program name, successor-list*⟩
A	⟨P_1, B⟩, ⟨P_2, C⟩
B	⟨P_1, NIL⟩
C	⟨P_2, NIL⟩
D	⟨P_3, E F⟩
E	⟨P_3, G H⟩
F	⟨P_3, E⟩
G	⟨P_3, NIL⟩
H	⟨P_3, NIL⟩

3.3 Using PBS

Probability-based prediction models have an intrinsic problem of slow adaptation. The same prediction is made until enough different events occur to change the corresponding probability. This can cause some problems in predicting the next file. Take the file A in Figure 1 for example, a probability-based method will predict B or C to be the next file depending on which one has followed A more often. A probability-based model does not realize that executing P_1 or P_2 is the reason behind the two different patterns, AB and AC. In other words, probability can only reveal *what* the result is, but not *why* it occurs. The situation can get worse as the number of programs accessing A increases.

Program-based Last Successor (PLS) is a specific instance of PBS where, for each program that has accessed a file, only the most recent program-based successor of that file is stored. Thus each file has a list of program-based successors, one per program that has accessed the file. When a program accesses a file, the last successor of that file for that program is predicted as the next successor of that file. Our simulation results show that it performs better than the well-known Last-Successor (LS) model and the recently developed Finite Multi-Order Context (FMOC) model.

4 PLS, LS, and FMOC Models

PLS is derived from PBS. It performs file prediction to reduce program stall time. In this section, we discuss why PLS performs better than LS. We also revisit FMOC which outperformed LS in previous study. The outcome of the comparison indicates that PLS can more accurately predict next needed file than the other two.

4.1 LS and FMOC

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.

FMOC predicts the next file to be accessed from the files that have been seen so far in "*context*" [9]. 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 [9]. 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" [9]. Since LS only predicts one file at a time, we add one to its score if it makes a correct prediction. No score is added for a wrong prediction. The final score is also normalized. 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).

4.2 PLS

As mentioned above, PLS incorporates knowledge about the running programs to generate a better last-successor estimate. More precisely, PLS records and predicts program-specific last successors for each file that is accessed.

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 2, the file access pattern AB is seen to

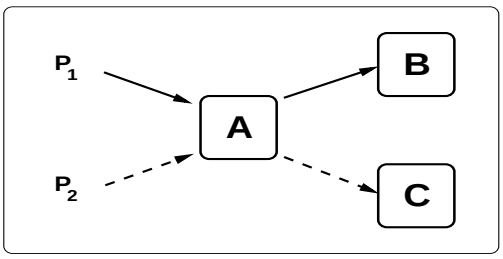

Figure 2: Program-based Last-Successor model

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.

Table 2: Metadata of Figure 2 kept under PLS model

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

One can argue that the same program may access different sets of files each time that it is executed, particularly a system utility program such as a compiler. While it is true that compiling different programs will result in different files being accessed, compiling the same program multiple times will result in many or all of the same files being accessed in the same order. Thus PLS will make correct predictions for most of these files, even when alternating compilations between two sets of files. Assume, for example, that two programs need to be compiled. The first program needs files X_1, X_2, ..., X_m, in that order, and the second program needs files Y_1, Y_2, ..., Y_n, in that order. If X_1 and Y_1 are different files, then we don't know which file to predict when the compiler starts running, but as soon as either X_1 or Y_1 is accessed we know which file to prefetch next. If X_1 and Y_1 are the same, then we prefetch this file and wait to see whether X_2 or Y_2 is needed, and then we can predict the next file after that. Hence we can predict all files except the first occurrence of $X_i \neq Y_i$ ($i \leq \min(m, n)$) until the access to the next shared file X_j (which is same as Y_j, $i < j$) comes up.

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

Two issues that need to be addressed are how to collect the metadata in terms of ⟨*program name, successor*⟩ for each file, and how big the metadata needs to be in order to make accurate predictions. The first issue is quite simple. Building PLS is similar to building PBS, except that for each file, PLS only tracks the most recent successor per program and does not keep all previous successors as does PBS. In the example of Figure 2, when P_1 accesses the next file (say B) after its access to A, we update the metadata of A with ⟨P_1, B⟩, and next time P_1 accesses A, PLS can predict that the next file accessed will be B. Similarly, A also keeps ⟨P_2, C⟩ as parts of its metadata. The metadata of files in Figure 2 is shown in Table 2.

The second issue is not quite as simple as the first. Ideally, for each file we would like to record the name of every program that has accessed it before, along with the program-specific successor to the file, so that we know which file 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, 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.

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 PLS 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.

5 Experimental Results

In the section, we will discuss the trace data we used to conduct our experiments, and how we compare performance of FMOC2, LS, and PLS.

5.1 Simulation Trace and Experimental Methodology

In examining PLS we used the trace data from *DFSTrace* used by the *Coda* project [7, 13]. These traces

were collected from 33 machines during the period between February of 1991 and March of 1993. We used data roughly equal to the second half of the entire trace 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. Table 3 lists the period of trace for each machine used in our simulation. Research has demonstrated that the average life of a file is very short [1]. Therefore, instead of tracking every *READ* or *WRITE* event, we track only the *OPEN* and *EXECVE* events in our simulation.

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

We score PLS the same way we score LS, by adding 1 for each correct prediction and 0 for each incorrect prediction. We normalize the final scores of PLS and LS 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 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 10 and 13 months), it turns out that the effect of this compulsory error is negligible and does not affect the prediction accuracy comparison among the models.

5.2 Model Comparison

We used the filtered trace data to evaluate FMOC2, LS, and PLS. Figure 3 shows that PLS has the highest predictive accuracy in all machines. For models predicting one or more files at a time such as FMOC2, the additive accuracy indicates the likelihood that the next file actually referenced is among those predicted files. However for models predicting one file each time, like LS and PLS, there is no difference between the additive accuracy and the predictive accuracy, which represents the percentage of the time that a prediction model correctly predicts the next file. The comparison of LS and PLS in Figure 3 is re-displayed in Figure 4 in terms of predictive accuracy. Note that the values are the same as for the additive accuracy shown in Figure 3.

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

Table 3: Trace data used

machines used	Barber	Mozart	Dvorak	Ives
begin month	4/92	3/92	6/92	6/92
end month	2/93	3/93	3/93	3/93
months covered	11	13	10	10

Clearly, in order to examine the real performance of a prediction model, we need other information besides predictive accuracy. Thus, we use LS as the baseline to evaluate the performance of PLS in three categories. The first category is the percentage of total predictions (including correct and incorrect predictions) made by PLS as compared with LS. This percentage should not be to too small, otherwise PLS may be an unrealistic model just like the model B above. The second is the percentage of correct predictions made by PLS as compared with LS. This number should be as high as possible. The last category is the percentage of incorrect predictions made by PLS as compared with LS. Ideally this percentage should be less than 100%, indicating that PLS makes fewer incorrect predictions than LS. However, a percentage of over 100% does not necessarily reflect worse performance. After all, the percentage of correct prediction is also a factor in deciding the overall performance.

5.3 Category Performance

We cannot do the same comparison with FMOC2 due to the nature of the FMOC model as discussed above. Figure 5 displays the performance in the category of total prediction. It shows that the percentage of events where a prediction was made by PLS is only about five percent less than the numbers of LS. This is close enough to consider PLS to be a practical prediction algorithm in terms of the number of predictions it makes. The percentage of correct predictions is shown in Figure 6. The percentage for Barber from PLS is over 99% of the number from LS, for

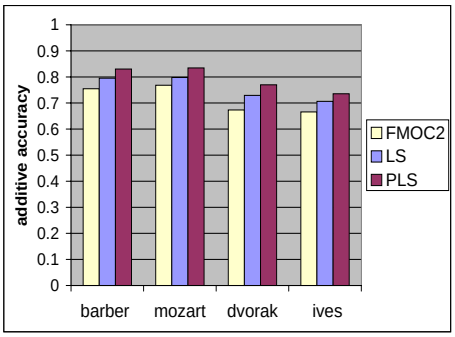

Figure 3: Additive accuracy of FMOC2, LS, and PLS

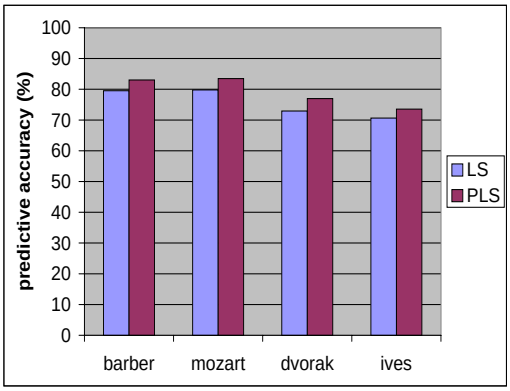

Figure 4: Predictive accuracy of LS and PLS

Ives it is over 98%, and for both Mozart and Dvorak PLS makes more correct predications than LS. Figure 6 demonstrates that PLS can do roughly as well as LS in correctly predicting files. Figure 7 shows that PLS makes fewer incorrect predictions than LS. Figure 8 shows the same data normalized by LS and shows that PLS indeed makes 15 to 22% fewer wrong predictions than LS, which is a very exciting result. This explains why PLS has the highest predictive accuracy among all three models 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 PLS is significant enough to be worthy of further exploration. Since the number of predictions made by PLS is only about five percent less than LS, and the number of correct predictions is roughly same as LS, we conclude that PLS makes no prediction more often than LS. We collected the percentage of cases where no prediction was made by PLS compared with LS, and the results are displayed in Figure 9, which confirms this surmise. Figure 9 shows that the percentage of events where no prediction was made by PLS is roughly three to six times higher than that of LS.

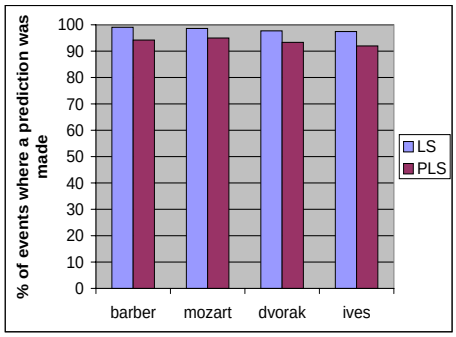

Figure 5: Total predictions made by LS and PLS

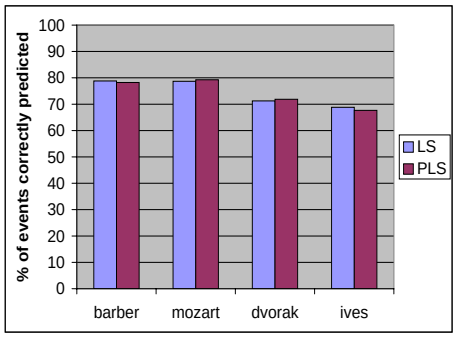

Figure 6: Correct predictions made by LS and PLS

We stated earlier that some events were filtered out of our trace data due to the requirement that PLS needs to know the program initiating an event, and we claimed that the filtering does not affect the validity of our results. To verify this, we compared the percentage of events filtered out of original trace data with PLS predictive accuracy for each machine. Our assumption was that if the filtered data had affected our results, the effect would be greater for larger amounts of filtered data. However, the results in Figure 10 show that the predictive accuracy of PLS (the back row) is unrelated to the percentage of events filtered

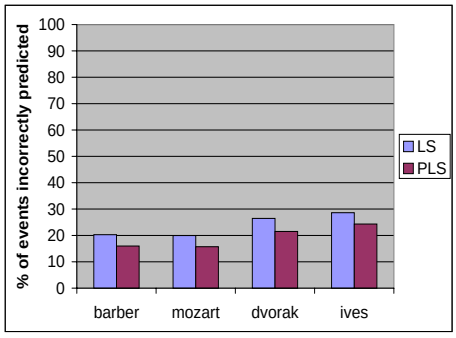

Figure 7: Incorrect predictions made by LS and PLS

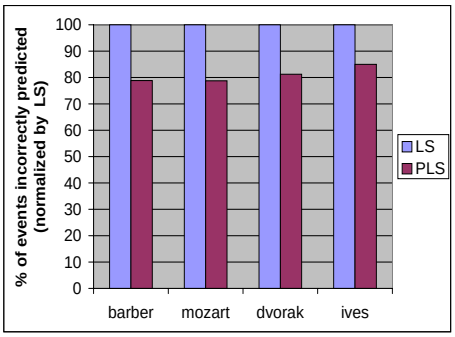

Figure 8: Incorrect predictions made by LS and PLS (normalized to LS)

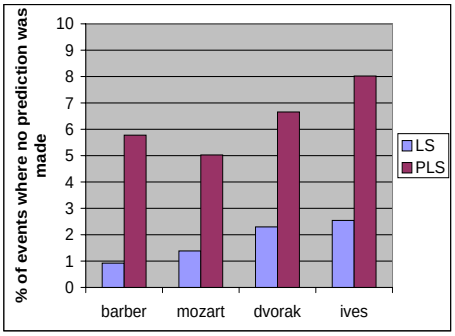

Figure 9: No predictions made by LS and PLS

out from the original trace data of each machine (the front row).

One last note about the number of ⟨*program name*, *successor*⟩ pairs that a file requires to successfully implement PLS. Our simulation results show that for Barber, more than 99% of files are accessed by six or fewer programs, while more than 99% of files are accessed by five or fewer programs for the other three machines. Thus the amount of data stored for each file in PLS is not of concern.

In addition to predictive accuracy we also want to know how PLS performs in terms of cache hit ratio, and additional experiments were conducted to determine this. We set the cache size according to the number of files it can hold for two reasons. The first is that file size is usually small, so the entire file can often be prefetched into cache [16]. The second is that in the case of large files, sequential read is the most common activity. Modern operating systems can already identify sequential read accesses and techniques such as prefetching the next several data blocks for sequential read have been implemented. We simulate cache with different sizes ranging from 25 files to 2000 files, and compare the cache hit ratios between the LRU caching algorithm with no prediction and

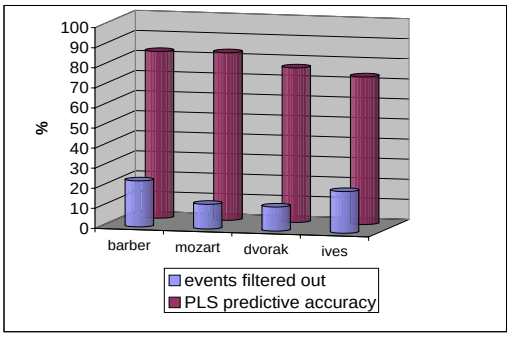

Figure 10: PLS performance vs. percentage of events filtered out of original trace data

the LRU caching algorithm with PLS. Figure 11 shows that when using PLS 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.

Part of the reason for this dramatic performance improvement is the fact that an incorrect prediction made by PLS, 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 PLS makes program-based predictions, its incorrect predictions are much more likely to be for a file to be accessed in the near future than are predictions made by non-program-based models, which may predict a file accessed by a program that is no longer even running. In other words, the incorrect predictions by PLS are more likely to be used in the near future and are therefore less wrong than those made by other models. The earlier graphs showing 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 11 clearly shows this effect. In real systems where multiple files can fit in memory at once, the performance will benefit accordingly.

6 Future Work

Several alternatives may improve the performance of PLS and are worthy of further exploration. For example, further classifying file accesses based upon the user for which the program is running may improve the performance of PLS. PLS may also use the preceding file together with the ⟨*program name*, *successor*⟩ to improve performance. This can resolve some ambiguous cases such as predicting next file after the access to file E in the example of Figure 1. On the other hand, files existing temporarily (such as those in */tmp* directory) usually

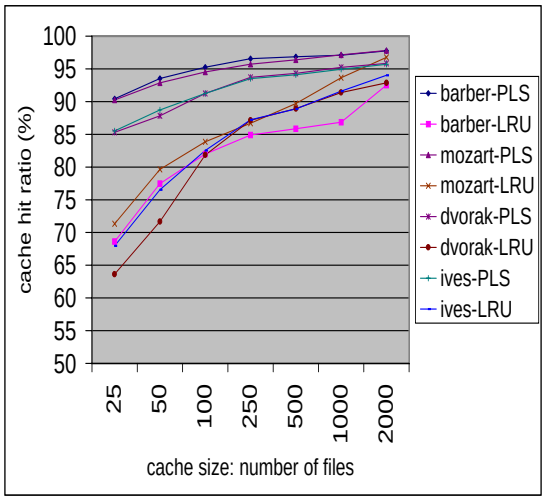

Figure 11: Cache hit ratio of LRU (alone) and LRU with PLS

won't get the same name next time they are created again. If so, then they can never be predicted correctly by PLS and there is no need to store their information.

Currently we are also examining the feasibility of applying the PBS model to conserve battery energy in mobile computers by prefetching multiple files each time and then spinning down the disk to save energy. This could greatly affect the utility of mobile computers since battery life is extremely valuable in a mobile environment. File hoarding is another area where we are considering the use of PBS and PLS. Files needed by programs can be preloaded over the network to the local disk so that programs can continue to execute when the network connection is unavailable.

7 Conclusions

As the speed gap between CPU and the secondary storage device will not be narrowing in the foreseeable future, file prefetching will continue to remain a promising way to keep programs from stalling while waiting for data from disk. We have developed a new program-based successor model, PBS, which efficiently stores file access histories and enables accurate file access predictions. Our simulations from PLS, derived from PBS, show good results in predicting files.

File accesses are driven by the programs using them, not by previous access patterns. Pattern history has been broadly used to predict the next needed object. A good predictor must adapt itself promptly as the access pattern changes. *Last-successor* is a good prediction model in general, but we have shown that it can be improved by applying it to program-based access patterns. By tracking programs initiating file accesses, we successfully avoid many incorrect predictions, as our results demonstrate. More than 21% of incorrect predictions can be reduced as compared with LS in some cases. Therefore, the overall performance penalty caused by incorrect predictions can be significantly reduced. We also compare the cache hit ratios of LRU with and without PLS. The results show that with PLS, LRU can deliver a much higher cache hit ratio.

8 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 R. Burns and E. Miller. Finally, we are grateful to the Coda group for providing us access to their traces.

References

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

[2] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of Integrated Prefetching and Caching Strategies. In *ACM SIGMETRICS*, 1995.

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

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

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

[6] Trace Kimbrel, Andrew Tomkins, Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Second Symposium on Operating Systems Design and Implementation*, 1996.

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

[8] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the first Parallel and Distributed Information Systems, IEEE*, 1991.

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

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

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

[12] Todd Mowry, Angela Demke, and Orran Krieger. Automatic Compiler-Inserted I/O prefetching for Out-of-Core Applications. In *The Second Symposium on Operating Systems Design and Implementation*, 1996.

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

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

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

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

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

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