

# Improving the System Performance by a Dynamic File Prediction Model

Tsozen Yeh, Joseph Arul,  
Kuo-Hsin Tien, I-Fan Chen, and Jia-Shian Wu  
Department of Computer Science and Information Engineering  
Fu Jen Catholic University, Taiwan  
{yeh, arul, kstan91, tick92, intel91}@csie.fju.edu.tw

## Abstract

*As the speed gap between CPU and I/O is getting wider and wider, I/O latency plays a more important role to the overall system performance than it used to be. Prefetching consecutive data blocks within individual files has been explored to hide, at least partially, the I/O latency. Unfortunately, this does not reduce the inter-file latency occurring when a program reads new files during its execution. We implemented a file prediction technique called Program-Based Successor (PBS), which effectively predicts and prefetches upcoming file requests for programs in execution. Inspired by the PBS model, we developed a dynamic prediction model called Dynamic Program-Based Successor (DPBS). DPBS dynamically adjusts the number of files prefetched according to the real-time system environment. Consequently, compared with PBS, it can further leverage the effectiveness of prefetching. We used multiple modified Andrew benchmarks to evaluate our DPBS system implemented in Linux kernel. The results show that the DPBS system can effectively reduce the elapsed time by up to 18%.*

**Keywords:** file prediction, prefetching

## 1. Introduction

Running programs stall if the data they need is not in memory. As the CPU speed dramatically increases in recent years, disk latency becomes more expensive in terms of the CPU cycles spent on waiting for the data to be read from disk. Consequently, disk I/O shows its tendency to bottleneck the overall system performance. Unfortunately, due to the mechanical nature of disk operations, it is unlikely to narrow the speed gap between CPU and hard drives in the foreseeable future. One way to mitigate the speed gap impact is to predict and prefetch files that running programs may need into main memory before they are actu-

ally accessed by programs in execution. Prefetching not only hides disk latency by overlapping between disk I/O and CPU operations, it also could benefit disk scheduling. When a disk has multiple requests to handle, its controller can find a better global approach to service all requests. Previously we developed a file prediction model called Program-Based Successor (PBS) [15]. Our simulation results showed that PBS can predict the next upcoming file request more precisely than some other state-of-the-art prediction techniques.

One might think that the success of file prediction solely depends on the accuracy of the file prediction algorithm – how accurately an algorithm can predict files needed. However, bringing files from disk to memory takes time, a correctly predicted file will be of no use if the system cannot preload it into memory before the time running programs need it. Therefore, making prediction only for the next upcoming file access may not deliver the expected results. On the other hand, preloading too many files into memory could have useful data swapped out of memory. Disk I/O channel can also be clogged if an excessive number of predicted requests constantly generated. To leverage the strength of prediction, we modified PBS to create a variation called *Dynamic Program-Based Successor* (DPBS) model, which dynamically adjusts the length of prediction sequence according to the real-time system situation. Unlike PBS, DPBS makes prediction for the next several file accesses according to the run-time free memory space left in the system. Since the system situation keeps changing, fixing the length of prediction sequence is apparently not appropriate. Previous prediction study also suggested that the outcome of file prediction varied greatly on the length of prediction sequence [7]. Generally speaking, systems with more free memory left can afford more predicted files.

We implemented both PBS and DPBS into Linux kernel. To evaluate our implementation, we used multiple instances of modified Andrew benchmark to conduct experiments. As expected, our experimental results showed that PBS can effectively reduce the elapsed time, and DPBS outperformed

PBS in all cases. In particular, DPBS reduced the elapsed time by up to 18%.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 overviews the PBS model. Section 4 details the implementation of the PBS and DPBS. Section 5 describes our experiments. Section 6 presents and evaluates our experimental results. Section 7 presents the conclusion. The future work is mentioned Section 8.

## 2. Related Work

Various schemes had been developed to make file-access prediction. Most of them used prior file access patterns to make prediction, while few obtained help from the compiler to do the work.

Kroeger and Long predicted the next upcoming access event based on the Partitioned Context Modeling (PCM) [8]. They later improved Partitioned Context Modeling (PCM) to Extended Partitioned Context Modeling (EPCM), which predicts sequences of files with observed probability higher than a given value [7]. Lei and Duchamp used pattern trees to record past execution activities of individual programs [9]. They maintain different pattern trees for every access pattern observed. A program could require multiple pattern trees to store similar patterns of file accesses observed from its previous execution, which imposes storing duplicated information in the system. Patterson *et al.* developed Transparent Informed Prefetching system to do prediction by using hints provided from modified compilers [13]. Accordingly, resources can be managed and allocated more efficiently. Extra coding in programs and language dependence are disadvantages of this type of approach. Griffioen and Appleton used probability graphs to predict future file accesses [3]. Their graph model tracks subsequent file accesses observed within a certain window for each file access. For a given file access, the observed followers with probability higher than a threshold value are the targets to prefetch. Their simulations show that the length of the window and probability threshold will greatly affect the outcome. Mowry *et al.* used modified compiler to provide future access patterns for out-of-core applications [11]. Kotz and Ellis defined representative parallel file access patterns in parallel disk systems [6]. Cao *et al.* defined four properties that an optimal predicting and caching model should satisfy [1]. Palmer and Zdonik used unit pattern to prefetch data in database applications [12]. Kimbrel *et al.* examined four related algorithms to find out when a prefetching algorithm should act aggressively or conservatively [5].

Researchers have found that files in the same directory often are accessed together [2, 10, 14]. Not surprisingly, some filesystems manage to put those files to disk areas where they can be collectively accessed more quickly [10].

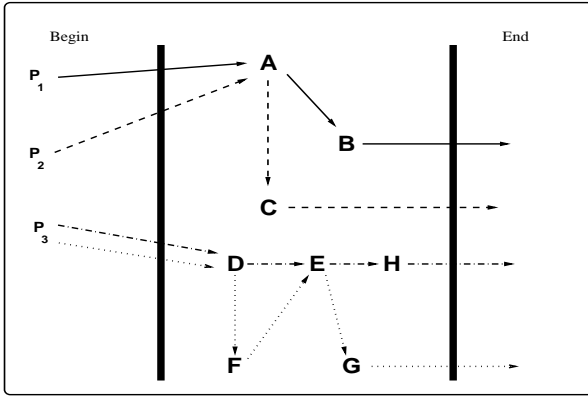
## 3. Program-Based Successor (PBS) Model

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  $P_2$ . 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.

The Program-Based Successor (PBS) model keeps track of previous executions of programs and the sequence of files they have accessed [15, 16]. 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  $G$ .

For each file PBS stores pairs of  $\langle \text{program name}, \text{successor-list} \rangle$  in the metadata of the file. The *program name* in each pair represents a program which accessed that



**Figure 1. a simple example of the program-based successor model**

file before. The *successor-list* is a list of files that the program accessed immediately after previous accesses to the corresponding file.

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  $\langle P, Y \rangle$ . 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 metadata now becomes  $\langle P, Y, Z \rangle$ . Table 1 shows the metadata kept for Figure 1 under the PBS model.

**Table 1. metadata of Figure 1 kept under the PBS model**

file	$\langle \text{program name}, \text{successor-list} \rangle$
A	$\langle P_1, B \rangle, \langle P_2, C \rangle$
B	$\langle P_1, NIL \rangle$
C	$\langle P_2, NIL \rangle$
D	$\langle P_3, E F \rangle$
E	$\langle P_3, G H \rangle$
F	$\langle P_3, E \rangle$
G	$\langle P_3, NIL \rangle$
H	$\langle P_3, NIL \rangle$

#### 4. Dynamic Program-Based Successor (DPBS)

We first implemented PBS as described in Section 3 into Linux kernel. We then modified the PBS to make consecutive prediction according to the length of prediction se-

quence. Finally, we added a monitor agent dynamically adjusting the length of prediction sequence to guide the modified PBS to make file-access prediction.

#### 4.1. Managing PBS Information

The Linux kernel has a layer of Virtual File System (VFS), which appears to be a reasonable place to do the PBS work. In VFS every directory and every file have a corresponding *d-entry* structure. Collectively, these *d-entry* structures form a tree which helps to find the full paths of directories and files. It seems that we can store PBS-related information in the *d-entry*. However, since the *d-entry* structure does not record the full path of its corresponding file that the PBS model must obtain, so the system needs to go through multiple *d-entry* structures to find out the full path of any file. We conclude that keeping PBS information in *d-entry* will complicate the implementation and probably will also slow the performance down. As a result, separate structures were built to handle PBS information. We created two structures, *File\_Successor\_Record* (FSR) and *File\_Successor\_Record\_Table* (FSRT). The FSR is used to record all the PBS information, while FSRT is used to maintain FSR.

#### 4.2. File Successor Record (FSR) and File Successor Record Table (FSRT)

As seen in Section 3, the information in Table 1 is required to build PBS system. The *File\_Successor\_Record* (FSR) is the structure storing PBS information regarding the programs, files, and successors, similar to those in Table 1 for individual files. To handle files or programs with the same name, the information recorded in FSR should include their full paths as well. Our previous study showed that the vast majority (between 87% and 96%) of files are accessed by three or fewer different programs [16]. Besides, most program-based successors don't vary often. Half of the time the program-based successors never get changed. About 10% to 15% of the total file accesses will see only two different program-based successors for a particular program, while 10% will observe three. So for each file, we keep the information for the most recent three programs, which had an access to the file before, and we record up to the last three different program-based successors for each of the three programs.

Each file (from *A* to *H*) in Table 1 has a corresponding FSR. A file's FSR is created when it is opened for the first time. It mainly includes the full-path (absolute path) name of the file, corresponding programs and successors, as well as a hash code (file-name hash code) generated from its name used to locate the FSR itself from the FSRT promptly. A file's program-based successors recorded in its FSR will

be updated accordingly if they ever changed.

In order to efficiently maintain and locate files' FSRs, we create an FSRT table to manage all FSRs in the system. Each FSR holds an entry in the FSRT table. A file's FSR can be indexed by a new hash code generated from its original file-name hash code. A link between a file's FSR and *d-entry* is established at the time the FSR is created. So the subsequent accesses to the file's FSR can be performed directly from its *d-entry* without going through the indexing again once the FSR is created. FSRs are stored in memory. Eventually the system needs to save them into disk to avoid losing the information. FSRT can be reconstructed, it does not need to be saved into disk. Currently our implementation writes FSRs to a disk file when the system shuts down and reads them back to memory as soon as the system comes up again next time. This can be easily modified to save FSRs to disk periodically (similar to the way UNIX handles its dirty blocks) or whenever the system is not busy. FSRs and FSRT only consume limited memory space. In our implementation, the FSRs and FSRT for every 8000 files require merely about 2 MB of memory space. Alternatively, we can put the FSR for each file into its inode (index node) when the memory space required to recording FSRs and FSRT becomes an issue.

### 4.3. Performing File Access Prediction

Figure 2 illustrates the process of performing file-access prediction and prefetching. When a file, say *X*, is opened by a program, say *P*, the system first finds the *X*'s corresponding *d-entry*. A file's *d-entry* mainly includes a pointer to its inode, links to its parent and sibling *d-entry*, and a hash code generated from its file name. After locating the *d-entry*, we take the hash code and hash it again (as step 1). We then use the new hash code as an index to the FSRT table to get the *X*'s FSR entry (as step 2). If the *X*'s FSR does not exist, this means that file *X* is opened for the first time. So we simply create an FSR entry for file *X* and no prediction will be made. However, if the *X*'s FSR exists, we will find the appropriate program-based successor if it exists. If that successor is not in memory yet, we will add it to the prefetch queue (as step 3), which our prefetch engine will check and send service requests to disk from. The prefetch engine is a daemon. It is scheduled to run after every file-access prediction made. When the system schedules the prefetch daemon to run, the daemon checks the prefetch queue and sends service requests to disk until either the queue is empty or it runs out of its time quota. We set a time quota to the prefetch daemon so it will not clog the disk I/O channel by prefetching an excessive number of files in case the prefetch queue is extremely long. As Linux automatically performs the block-level read ahead, our prefetch daemon only requests the first block of the prefetched file.

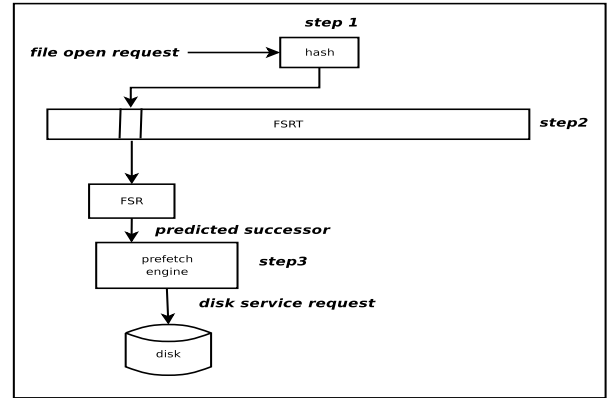


Figure 2. the process of performing file-access prediction and prefetching

### 4.4. Monitor Agent

There are two issues we need to consider when designing the monitor agent. First, how often the agent adjusts the length of prediction sequence. The agent was implemented through a kernel thread, which has a very high priority. Potentially it could aggressively withhold the CPU from other running programs. We can prevent this problem by waking up the agent periodically, or more dynamically, we wake it up after the prefetching engine finishes sending its prefetching requests and then we make the agent sleep right after each length adjustment. Our experimental results showed that the dynamic way outperformed the periodic way.

The second issue is how much adjustment the monitor agent makes to the length of prediction sequence each time. Between predefined high and low watermarks, the agent increases or decreases the length by one as the free memory space grows or shrinks every 4% of the overall memory space respectively. Compared with the overall memory space, free memory space is viewed on a scale of 1 to 25, so level 1 means only 4% of overall memory space is free, while level 25 represents the memory space is 100% free (of course memory space will not be 100% free in reality). We tried different scales and found that the results among different scales are very close as the scale reaches 1 to 25 or above. However, once the scale goes down to 1 to 20 or below, the outcome started to fall. This suggests that scale granularity should be considered when the monitor agent makes the length adjustment.

As stated, the timing to execute the monitor agent can be either periodical or dynamic. We used ten different periods, from every one second to every ten seconds, in our experiments. The one-second period means the monitor agent wakes up once every second, while the ten-second period is the case that the agent wakes up once every ten seconds.

Table 2 lists all agents with their individual wake-up schedules. The agent “Xs” represents the agent waking up with the  $X$ -second period. The agent  $D$  dynamically wakes up and sleeps as we discussed earlier.

**Table 2. agents used in the experiments**

Agents	Wakeup Period
1s	every one second
2s	every two seconds
3s	every three seconds
4s	every four seconds
5s	every five seconds
6s	every six seconds
7s	every seven seconds
8s	every eight seconds
9s	every nine seconds
10s	every ten seconds
D	same as the prefetch engine

## 5. Experiments

We first present the results from tests on our implementation of the PBS and DPBS models. We then explain how various agents with different wake-up schedules perform under the DPBS model. Our test machine had a Pentium-4 2.4G-Hz CPU, with 512 MB RAM, an IDE Maxtor 7200RPM disk with DMA66. All kernels were compiled without symmetric multi-processor(SMP) support. The system used GNU ld version 2.14.90.0.8, and gcc version 3.3.4. We implemented the DPBS into Linux kernel 2.4.22.

We used the Andrew benchmark [4] to evaluate our implementation. In addition, we also want to see how our model works when the system hosts more programs and intensive disk I/O. So we simultaneously ran multiple instances (one, two, and four, all executed in different directories) of Andrew benchmark to simulate variant busy degrees of the system.

All test cases were repeated five times. We cleared all caches by rebooting the system before each run of a test case. As discussed later, the Andrew benchmark consists of five phases. Each one of the five launches a new program, so there is no way to make predictions and prefetch files before our models see their execution at least once. Therefore, the first run of a test case is used to train the model. The results presented are the average numbers from experiments in the remaining four times.

The training cost of the DPBS is low. We only saw about three percent performance penalty. The effectiveness of our model is presented in terms of improvement. We calculated how much elapsed time our implementation saved over the original kernel not making prediction. The difference was

**Table 3. phases of the Andrew benchmark**

Phase	Command
1	/bin/mkdir
2	/bin/cp
3	/bin/stat
4	/bin/grep
5	/usr/bin/gcc

then divided by the original elapsed time to get the percentage of improvement.

One might concern the cost of re-training a file prediction model, even when a program only slightly changes the sequence of files it accessed before. For the DPBS model, it just needs to add the new program-based successor to the corresponding FSRs. In other words, the three-percent performance penalty occurs only if programs completely change their accessing orders of every file whenever they execute. Our previous long-term (over a period of one year) trace-based simulation showed that this rarely happened [15]. So the long-term cost of maintaining the DPBS model will be much lower than the three-percent performance penalty. If we add the cost of maintaining the DPBS model and the benefit of performing file prefetching, we can clearly see the advantage of using this model.

### 5.1. Andrew Benchmark

The Andrew benchmark consists of five phases. Each one of the five launches a UNIX command as listed in Table 3. One may apply these five phases to different testing files. The first phase creates directories, while the second phase copies the files to those directories. The third step reads inodes of those files to get their file status. The *grep* in the fourth phase searches files for patterns. The last phase compiles files to object files and link them into one executable file. We used the source code of *Tcl* in our experiments. The *Tcl* source includes 76 files written in the *C* language.

## 6. Performance Evaluation

To evaluate how our models perform under various busy degrees of system and disk I/O, we conducted experiments with multiple instances (one, two, and four) of modified Andrew benchmark running concurrently. For simplicity, we will refer to the case concurrently running  $X$  instances of Andrew benchmark as  $X$  Andrew in the following discussion. We collected the original elapsed time of running Andrew benchmark without making prefetching, as well as the elapsed time of our models. Table 4 lists the percentage of elapsed time spent on each of the five phases as running one instance of Andrew benchmark without making prefetching.

**Table 4. percentage of elapsed time in the five phases of the Andrew benchmark**

Phase	Percentage
1	0.15%
2	0.9%
3	0.15%
4	4.91%
5	93.98%

### 6.1. Performance of DPBS

Figure 3 shows the results for the case of running one instance of Andrew benchmark under the PBS and DPBS model. The  $X$  axis represents agents evaluated. The one labeled  $P$  stands for the PBS model. The agent  $D$  is the agent dynamically wakes up and sleeps. Agent “ $Xs$ ” denotes the one wakes up once every  $X$  second. For simplicity, we refer to these “ $Xs$ ” agents as *periodic* agents. The dynamic agent  $D$  did a little bit better than PBS in this experiment.

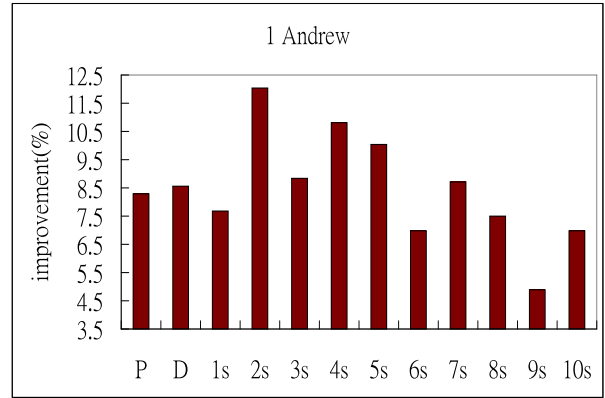
Figure 4 displays the results for the case of running two instances of Andrew benchmark currently. Compared with *1 Andrew*, this experiment simulates the situation where the system hosts more running programs and handles more disk I/O requests. DPBS with agent  $D$  outperformed PBS to a larger degree in this case. The performance of periodic agents still varied. Besides, agent  $D$  also had the best performance in *2 Andrew*.

Figure 5 plots the results from the *4 Andrew*, where the system environment is busier than the previous two. DPBS with agent  $D$  exceeded PBS further, and its result apparently stood out among all agents in *4 Andrew*.

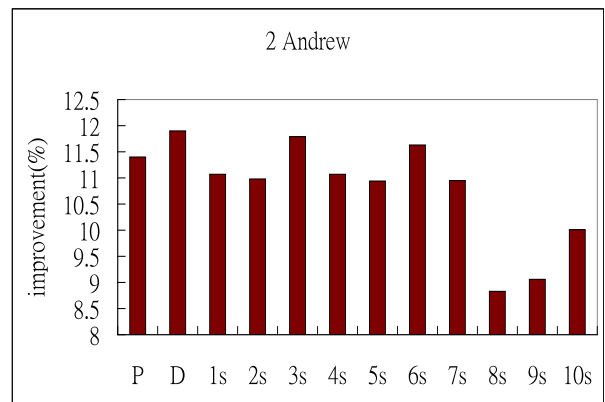
To compare PBS and DPBS in a more close way, we plot all experimental results into Figure 6. It shows that, with the dynamic agent  $D$ , DPBS always performed better than PBS, in particular for a system hosting more programs and I/O requests. However, we did not observe similar situations for those periodic agents. In fact, not surprisingly, the results from those periodic agents fluctuated. Since the real-time system situation changes quickly, agents waking up periodically to adjust the length of prediction sequence are not likely to generate consistent improvement. This suggests that there is no simple relation between different periodic agents and their outcome. Therefore, we should dynamically, including degree and timing, adjust the length of prediction sequence.

## 7. Conclusions

As the speed gap between CPU and the secondary storage device will not be narrowing in the foreseeable future,



**Figure 3. PBS vs. DPBS under the case of 1 Andrew**



**Figure 4. PBS vs. DPBS under the case of 2 Andrew**

file prefetching will continue to remain a promising way to keep programs from stalling while waiting for data from disk. We devised a dynamic file-access prediction model, DPBS. We believed that prefetching only for the next upcoming predicted file may not leave enough time for the file to reach memory in time. One solution is to prefetch multiple consecutive files, not just one file, whenever possible. A file-access prediction model should smartly adjust the length of prediction sequence to consistently deliver good performance. Nevertheless, the dynamic prediction is complicated by the fact that the amount of free memory space available is always changing. Consequently, it is not easy for a real prediction model to keep its performance improvement all the time. We implemented PBS and DPBS models into Linux kernel and conducted extensive experiments to show that the length of prediction sequence has an impact on the system performance. By dynamically adjusting the length of prediction sequence, our DPBS model delivers very dependable performance.

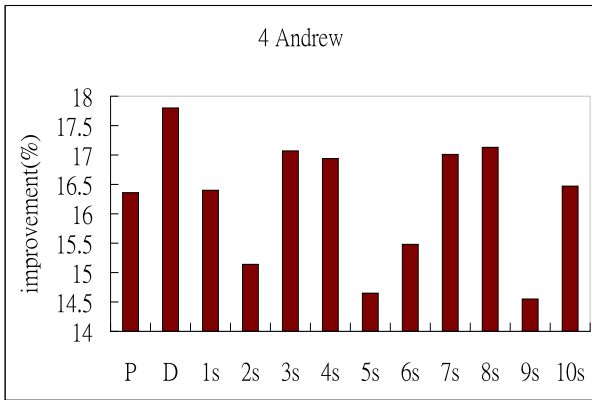


Figure 5. PBS vs. DPBS under the case of 4 Andrew

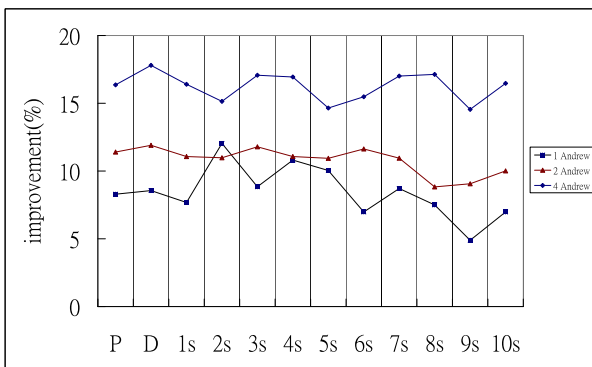


Figure 6. the performance improvement of the PBS and DPBS models

## 8. Future Work

Besides free memory space left, the number of pending I/O requests could also be used to improve a file prediction model. Ideally, fewer predictions should be made if there is a long list of pending I/O requests. A file prediction model may work with disk drivers to obtain the real-time situation in the disk level to make prediction adjustments. We did not take the advantage of exploring pending I/O requests in our implementation of PBS and DPBS. We may improve our DPBS model by considering the situation of pending I/O requests in the future.

## 9 Acknowledgments

This research is supported in part by National Science Council (NSC) award number 92-2213-E-030-008. We are grateful to the support from NSC.

## References

- [1] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of Integrated Prefetching and Caching Strategies. In *ACM SIGMETRICS*, 1995.
- [2] J. R. Douceur and W. J. Bolosky. A Large Scale Study of File-System Contents. In *Proceedings of the ACM SIGMETRICS*, 1999.
- [3] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. In *Proceedings of the Parallel and Distributed Computing System, IEEE*, 1995.
- [4] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. In *Transaction on Computer Systems*, 1988.
- [5] 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*, 1996.
- [6] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the first Parallel and Distributed Information Systems, IEEE*, 1991.
- [7] T. M. Kroeger and D. D. Long. Design and Implementation of a Predictive File Prefetching Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, 2001.
- [8] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.
- [9] H. Lei and D. Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX Annual Technical Conference*, 1997.
- [10] S. L. Marshall McKusick, William Joy and R. Fabry. A Fast File System for UNIX. In *Proceedings of the ACM Transactions on Computer Systems*, 1984.
- [11] 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*, 1996.
- [12] 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*, 1991.
- [13] 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*, 1995.
- [14] M. Rosenblum and J. Ousterhout. The LFS Storage Manager. In *Proceedings of the Summer 1990 USENIX Technical Conference*, 1990.
- [15] T. Yeh, D. 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)*, 2001.
- [16] T. Yeh, D. D. Long, and S. Brandt. Increasing Predictive Accuracy through Limited Prefetching. In *Proceedings of the Communication Networks and Distributed System Modeling and Simulation Conference*, 2002.