UC Santa Cruz

UC Santa Cruz Previously Published Works

Title

Conserving Battery Energy through Making Fewer Incorrect File Predictions

Permalink

https://escholarship.org/uc/item/5st644ps

Authors

Yeh, Tsozen

Long, Darrell

Brandt, Scott A

Publication Date

2001-05-01

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Conserving Battery Energy through Making Fewer Incorrect File Predictions

Tsozen Yeh, Darrell D. E. Long and Scott A. Brandt
Jack Baskin School of Engineering
University of California, Santa Cruz

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. Battery is a valuable resource in a mobile computing environment. The utility of mobile computers is greatly affected the battery life. Incorrect prediction not only wastes cache space, also consumes battery energy. Consequently, incorrect prediction is more expensive to mobile computers than its counterpart to desktop computers. Last Successor (LS) is a commonly-used predicting algorithm in practice. We present the *Program-based Last Successor* (*PLS*) file prediction model that identifies relationships between files through the names of the programs accessing them. Our simulation results show that PLS makes 21% fewer incorrect predictions and roughly the same number of correct predictions as the last-successor model. Hence, the amount of battery energy wasted on incorrect prediction could be greatly reduced. We finally examine the cache hit ratio of 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. This shows that saving battery energy and increasing performance can be reached at the same time.

1 Introduction

Running programs stall if the data they need is not in memory. As the speed of *CPU* increases, disk I/O becomes more expensive in terms of CPU cycles. File prefetching is a technique that mitigates the speed difference originating from the mechanical operation of disk and the electronic operation of CPU [20] by preloading files into memory before they are needed. 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 history of file access have been widely used to perform file prediction [3, 8, 12–14, 17], as have hints or help from programs and compilers [2, 18].

While correct file prediction is useful, incorrect prediction is to a certain degree both unavoidable and costly, particularly under a mobile environment where battery energy is considered as a critical resource. An incorrect prediction is much 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 cache space and battery energy of mobile computers. 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 and waste battery energy regardless of the accuracy of correct prediction. Therefore, reducing the number of incorrect predictions is very important to the utility of mobile computers.

We propose a new file prediction model, Program-based Last Successor (PLS), inspired by the observation that probability and repeated history of file accesses do not occur for no reason. We contend that the reason is that programs access more or less the same set of 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. PLS uses this knowledge to determine program-specific last-successors for each file to generate more accurate file predictions. Our results demonstrate that that PLS generates more accurate file predictions than the other file prediction algorithms examined. In particular, compared with LS, 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. The amount of battery energy wasted on incorrect prediction could also be reduced accordingly.

We compare PLS with LS and *Finite Multi-Order Context* (FMOC) [12]. 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 and other predicting algorithms in a one-month trace in Kroeger's study [12] 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. Consequently, compared with LS, PLS can reduce a large amount of battery energy wasted on incorrect prediction. 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.

Research has shown that appropriately spinning down the hard disk can greatly save battery energy [4–7, 9, 11, 15,22]. Most of energy-saving algorithms make disk spin-down decisions based on previous disk access pattern. Incorrect prediction could mislead these algorithms and lower their effectiveness by changing the original disk access pattern generated by running programs. PLS reduces many cases of incorrect prediction done by LS, which leads to a more accurate sequence of disk accesses closer to what really generated by programs in execution. In a real environment where file prediction is performed, energy-saving algorithms will benefit from this accordingly.

2 Related Work

Griffioen and Appleton use probability graphs to predict future file accesses [8]. 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. Lei and Duchamp use pattern trees to record past execution activities of each program [14]. They maintain different pattern trees for each different accessing pattern observed. Vitter, Curewite, and Krishnan adopt the technique of data compression to predict next required page [3, 21]. Kroeger and Long predict next file based on probability of files in contexts of FMOC [12]. Patterson *et al.* develop *TIP* to do prediction using hints provided from modified compilers [18]. Chang and Gibson design a tool which can transform UNIX application binaries to perform speculative execution and issues hints [2].

Greenawalt models disk accesses by a Poisson distribution [7]. Krishnan *et al.* [11] develop a two-stage rent-to-by algorithm to predict the period between current and next disk accesses. The first stage generates a small number of candidate periods, and the second stage chooses the candidate performing best as if it had been used in the past. Helmbold *et al.* [9] adopts a machine learning algorithm to generate certain number of fixed time-out periods as *experts* to predict the next time-out. Each expert is weighted by its current performance, and the weighted average of all experts is the predicted next time-out.

Douglis *et al.* [4] describe the undesirable waiting period as *bump* if the spin-up delay exceeds a certain percentage of the time that disk stays in spin-down status. They adaptively adjust the spin-down threshold depending on if the most recent spin-up delay is viewed as a bump or not. Li *et al.* [15] and Douglis *et al.* [5] demonstrate that using a shorter threshold in seconds instead of minutes commonly suggested by manufactures will save a large amount of energy.

3 LS, FMOC, and PLS Models

We start with a brief discussion of LS and FMOC models, followed by details of how to implement PLS model.

3.1 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*" [12]. 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 [12]. 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" [12]. 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

Figure 1: Program-based Last-Successor model

negligible improvements so in this work we only examine the second order FMOC model (denoted as FMOC2).

3.2 PLS

Lacking *a priori* knowledge of file access patterns, many file prediction algorithms use statistical analysis of past file access patterns to generate predictions about future access patterns. One problem with this approach is that executing the same set of programs can produce different file access patterns even if the individual programs always access the same files in the same order. 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.

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. 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 1, the file access pattern AB is seen to be caused by program P_1, while the file access pattern AC is caused by program P_2. In other words, what is really behind the numbers 60% and 40% is the execution of two different applications, P_1 and P_2. After we collect this information (a set of pairs consisting of "*program name*" and "*successor*") for file A, next time it is accessed we can predict either B or C depending on P_1 or P_2 is accessing A, or provide no prediction if A is accessed by another program. Of course, if a particular program accesses multiple different files after each access of a particular file, then the program-specific last successor will change.

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.

Table 1: Metadata of Figure 1 kept under PLS model

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

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 simple. Programs are executed as processes, so we can just store the *program name* in the process control block (*PCB*). For each running program (say P), we also need to keep track of the file (say X), which it has most recently accessed. When P accesses the next file (say Y) after X, we update the metadata of the X with ⟨P, Y⟩, and the next time that P accesses X, PLS can predict that the next file accessed will be Y.

In the example of Figure 1, when P_1 accesses the next file (say B) after its access to A, we update the metadata of A with $\langle P_1, B\rangle$, and next time P_1 accesses A, PLS can predict that the next file accessed will be B. Similarly, A also keeps $\langle P_2, C\rangle$ as parts of its metadata. The metadata of files in Figure 1 is shown in Table 1.

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.

4 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.

4.1 Simulation Trace and Experimental Methodology

In examining PLS we used the trace data from *DFSTrace* used by the *Coda* project [10, 16]. 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 2 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 one for each correct prediction and zero 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.

4.2 Model Comparison

We used the filtered trace data to evaluate FMOC2, LS, and PLS. Figure 2 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 regular predictive accuracy, which represents the percentage of the time that a prediction model correctly predicts the next file.

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.

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.

Table 2: 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

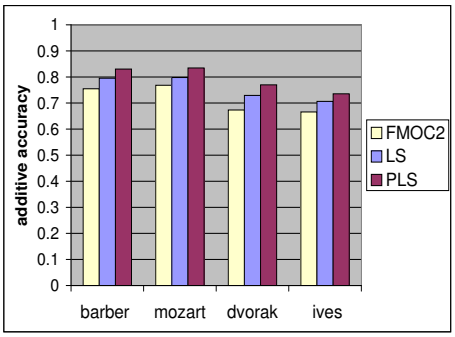

Figure 2: Additive accuracy of FMOC2, LS, and PLS

4.3 Category Performance

We can not do the same comparison with FMOC due to the nature of the FMOC model, as discussed above. Figure 3 displays the PLS performance normalized by LS in the three different categories. The columns marked "*total*" show that the total number of predictions made by PLS is about 95% of the number made by LS. This is close enough to consider PLS to be a practical prediction algorithm in terms of the number of predictions it makes. The middle columns marked "*correct*" are the percentages of correct predictions. The percentage for Barber from PLS is over 99% of the number from LS, for Ives it is over 98%, and for both Mozart and Dvorak PLS makes more correct predications than LS. Percentages from the middle columns demonstrate that PLS can do roughly as well as LS in correctly predicting files. Finally, the columns marked "*incorrect*" show that PLS indeed makes about 15% to 22% fewer wrong predictions as compared with LS, which is a very exciting result. This explains why the PLS model has the highest prediction accuracy among all three models in Figure 2.

The reduction of incorrect predictions in PLS is significant enough to be worthy of further exploration. Since the percentage of total predictions made by PLS is about 95% of LS, and the number of correct predictions is roughly same as LS, we conclude that PLS must make more no predictions than LS. We collected the percentage of no predictions from PLS compared with LS, and the result is displayed in Figure 4, which confirms this surmise. Figure 4 shows that the number of no predictioins made by PLS

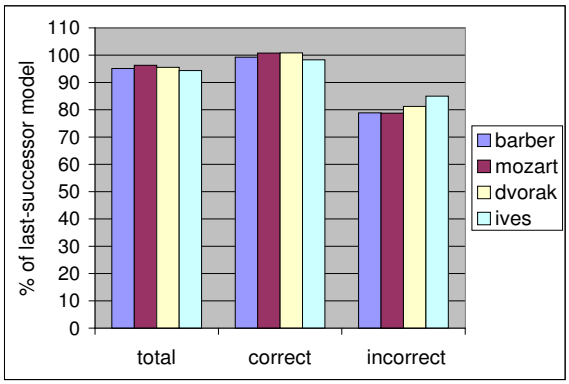

Figure 3: PLS performance normalized by LS in 3 separate categories

is roughly about three to six times more than that made by LS.

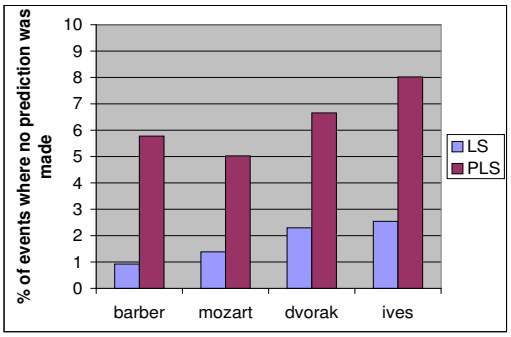

Figure 4: No predictions made by LS and PLS

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 5 show that the predictive accuracy of PLS (the back row) is unrelated to the percentage of events filtered out from the original trace data of each machine (the front row).

One last note about the number of $\langle$*program name*, *successor*$\rangle$ 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.

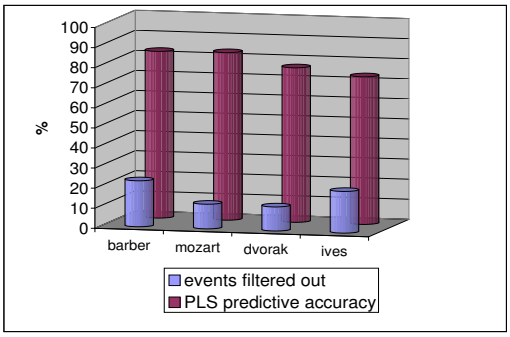

Figure 5: PLS performance vs. percentage of events filtered out of original trace data

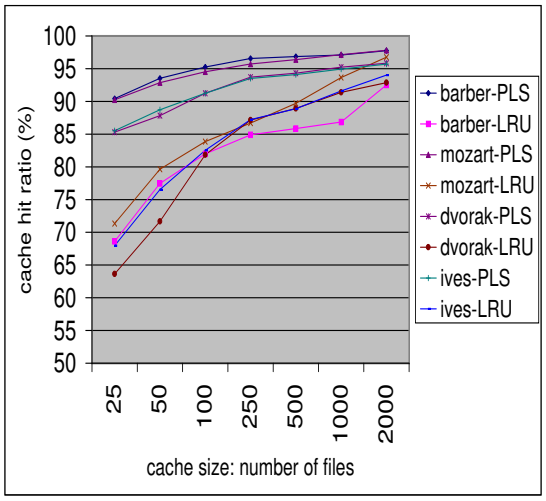

Figure 6: Cache hit ratio of LRU (labelled LRU) and LRU with PLS (labelled PLS)

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 [19]. 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 the LRU caching algorithm with PLS. Figure 6 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.

5 Future Work

Several alternatives may improve the performance of PLS and are worthy of further exploration. For example, files existing temporarily (such as those in */tmp* directory) usually 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. PLS may also use the preceding file together with the $\langle$*program name*, *successor*$\rangle$ to improve performance.

6 Conclusions

While file prefetching algorithm could improve performance, however unavoidable incorrect prediction will waste valuable battery energy of mobile computers. Reducing the number of files incorrectly predicted is very important to the utility of mobile computers in terms of saving both cache space and battery energy. We propose PLS, a new program-based last successor model. Our simulations from PLS show good results in predicting files, especially in eliminating the cases of incorrect prediction. More than 21% of incorrect predictions can be reduced as compared with LS in some cases as our results demonstrate. Hence, the amount of battery energy wasted on incorrect prediction can be reduced accordingly.

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

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

[4] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the Second Usenix Symposium on Mobile and Location-Independent Computing*, 1995.

[5] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *Proceedings of the Usenix Technical Conference*, 1994.

[6] Richard Golding, Peter Bosch, and John Wilkes. Idleness is not a sloth. Technical report, Hewlett-Packard Laboratories, 1995.

[7] Paul Greenawalt. Modeling power management for hard disks. In *Proceedings of the Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1994.

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

[9] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A Dynamic Disk Spin-down Technique for Mobile Computing. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, 1996.

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

[11] P. Krishnan, Philip Long, and Jeffrey Scott Vitter. Adaptive disk spin-down via optiman rent-to-buy in probabilistic environments. In *Proceedings of the Twelfth International Conference on Machine Learning (ML95)*, 1995.

[12] 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.

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

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

[15] Kester Li, Roger Kumpf andPaul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceeding of the Usenix Technical Conference*, 1994.

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

[17] 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.

[18] 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.

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

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

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

[22] John Wilkes. Predictive power conservation . Technical report, Hewlett-Packard Laboratories, 1992.