Predicting Future File-System Actions From Prior Events

Tom M. Kroeger and Darrell D. E. Long
University of California, Santa Cruz

Predicting File System Actions from Prior Events

Thomas M. Kroeger[†]and Darrell D. E. Long[‡§]
Department of Computer & Information Sciences
University of California, Santa Cruz

Abstract

We have adapted a multi-order context modeling technique used in the data compression method *Prediction by Partial Match* (PPM) to track sequences of file access events. From this model, we are able to determine file system accesses that have a high probability of occurring as the next event. By prefetching the data for these events, we have transformed an LRU cache into a predictive cache that in our simulations averages 15% more cache hits than LRU. In fact, on average our four-megabyte predictive cache has a higher cache hit rate than a 90 megabyte LRU cache.

1 Introduction

With the rapid increase of processor speeds, file system latency is a critical issue in computer system performance [14]. Standard *Least Recently Used* (LRU) based caching techniques offer some assistance, but by ignoring any relationships that exist between file system events, they fail to make full use of the available information.

We will show that many of the events in a file system are closely related. For example, when a user executes the program **make**, this will often result in accesses to the files **cc**, **as**, and **ld**. Additionally, if we note an access to the files **make** and **makefile** then another sequence of accesses: **program.c**, **program.h**, **stdio.h**, . . . , is likely to occur. As a result, the file system behaves predictably. Thus a predictive caching algorithm that tracks file system events and notes predictive sequences can exploit such sequences by preloading data before it is required. This increases the cache hit ratio and reduces file system latency.

As in data compression, where a model drives a coder, our predictive cache can be divided into two portions: the model that tracks the sequences of previous events and the selector that uses this information to determine likely future events and prefetch their data. Our model tracks previous file system events through a finite multi-order context modeling technique adapted from the data compression technique *Prediction by Partial Match* (PPM) [2]. This model uses a trie [9] to store sequences of previous file system events and the number of times they have occurred. Our selector examines the most recently seen sequences and the counts of the events that have followed them to determine likely next events. Using these predictions, we augment an LRU cache by prefetching data that are likely to be accessed. The result is a predictive caching algorithm that in our simulations averaged hit ratios better than an LRU cache that is 20 times its size.

The rest of this article is organized as follows: §2 details the method used to model events and select events to prefetch, §3 presents our simulations and results, §4 describes related work, §5 discusses future work, and §6 concludes the paper.

2 Predictive Caching Method

The problem of tracking file system events and the sequences in which they occur is quite similar to the text compression problem of tracking strings of char-

[†] Internet: tmk@cse.ucsc.edu, Telephone (408) 459-4458.
[‡] Internet: darrell@cse.ucsc.edu, Telephone (408) 459-2616.
[§] Supported in part by the Office of Naval Research under Grant N00014–92–J–1807.

acters to model their frequency distributions. The use of data compression modeling techniques for prefetching in operating systems was first investigated Vitter, Krishnan and Curewitz [19, 4]. It was their work that inspired us to adapt PPM's modeling techniques, to track file system events instead of characters of an alphabet. Using the information tracked by our model we chose a method based on a likelihood threshold to select the events to prefetch.

2.1 Context Modeling

Just as a word in a sentence occurs in a context, a character in a string can be considered to occur in a context. For example, in the string "**object**" the character "**t**" is said to occur within the context "**objec**". However, we can also say that "**t**" occurs within the context "**c**", "**ec**", "**jec**", and "**bjec**". The length of a context is termed its *order*. In the example string, "**jec**" would be considered a third order context for "**t**". In text compression these contexts are used to model which characters are likely to be seen next. For example, given that we have seen the context "**object**" it may be likely that the next character will be a space, or possibly an "**i**", but it is unlikely that the next character is an "**h**". On the other hand, if we only consider the first order context, "**t**", then "**h**" is not so unlikely. Techniques that track multiple contexts of varying orders are termed *Multi-Order Context Models* [2]. To prevent the model from quickly growing beyond available resources, most implementations of a multi order context model limit the highest order tracked to some finite number (m), hence the term *Finite Multi-Order Context Model*.

At every point in the string, the next character can be modeled by the last seen contexts (a set of order 0 through m). For example, take the input string "**objec**" and limit our model to a third order ($m = 3$). The next character can now be described by four contexts $\{$ ø, "**c**," "**ec**," "**jec**" $\}$. This set of contexts can be thought of as the current state of whatever we are modeling, be it a character input stream or a sequence of file system events. With each new event, the set of new contexts is generated by appending the newly seen event to the end of the contexts that previously modeled our state. If the above set was our current state at time t, and at time $t + 1$ we see the character "**t**", our new state is described by the set $\{$ ø, "**t**," "**ct**," "**ect**" $\}$. The nature of a context model, where one set of contexts is built from the previous set, makes it well suited for a trie[1] [9], where the children of each node indicate the events that have followed the sequence represented by that node. A resulting property of this trie is that the frequency count for each current context is equal to the sum of its children's counts plus one[2]. It is from this property that we derive our probability estimate in §2.3.

2.2 Tracking File System Events

In our model contexts are sequences of file system events. To store all the previously seen contexts we use a trie. Each node in this trie contains a specific file system event. Through its path from the root, each node represents a sequence of file system events, or a context, that has been previously seen. Within each node we also keep a count of the number of times this sequence has occured.

To easily update our model and use it to determine likely future events, we maintain an array of pointers, 0 through m, that indicate the nodes which represent the current contexts (C_0 through C_m). With each new event A, we examine the children of each of the old C_k, searching for a child that represents the event A. If such a child exists, then this sequence (the new C_{k+1}) has occurred before, and is represented by this node's child, so we set the new C_{k+1} (the $k + 1^{\underline{th}}$ element of our array) to point to this child and increment its count. If no such child is found, then this is the first time that this sequence has occurred, so we create a child denoting the event A and set the $k + 1^{\underline{th}}$ element of our array to point to its node. Once we have updated each context in our set of current contexts, we have a new state that describes our file system. Figure 1 extends an example from Bell [2] to illustrate how this trie would develop when given the sequence of

[1] A trie is commonly used as as a efficient data structure to hold a dictionary of words. It is based on a tree in which each node contains a specific character. Each node then represents the sequence of characters that can be found by traversing the tree from the root to that node.

[2] Note that since nodes of order m must have children (which will be leafs), a model of order m requires a trie of order $m + 1$.

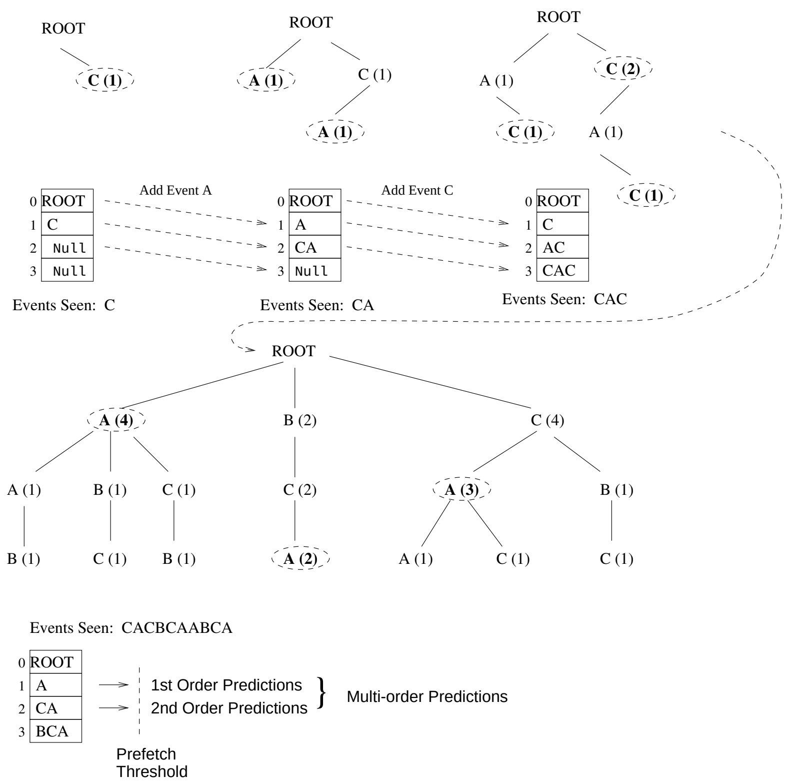


Figure 1: Example tries for the sequence $CACBCAABCA$.

events *CACBCAABCA*. The first three tries show how our model builds from the initial (empty) state. The last trie shows how our model would look after the entire sequence. The current contexts at each stage are indicated by the circled letters.

2.3 Selecting Events to Prefetch

As we generate each of the new contexts, we examine their children to determine how likely they are to be the next event. Using the formula $Count_{Child}/(Count_{Parent} - 1)$ we generate a maximum-likelihood estimation [17] of the probability of that child's event occurring. We compare this estimate to a probability threshold set as a parameter of our algorithm. If the estimated likelihood is greater than or equal to this threshold, then the data accessed for this event is prefetched into the cache. We evaluate each context 1 through m independently, resulting in m sets of predictions. The zero order context is a Least Frequently Used (LFU) model and therefore was thought to be of little benefit. Consequently the selector does not examine the zero order context for predictions.

Prefetched data is placed at the front of our cache, and since cache replacement is still LRU, the data will most likely be in the cache for the next several events. The result is that although our cache prefetches based on predictions for what the next event will be, since the prefetched data is likely to be in the cache for more than just the next event, as long as the event occurs before its data is removed from the cache we still avoid a cache miss.

3 Simulation

To simulate the workload of a system, we used file open events from the Sprite file system traces [1]. We chose to consider whole file caching for three reasons. The primary purpose of our work is to avoid the latency of file system accesses; if a whole file can be cached, this reduces the number of transactions with the I/O subsystem on behalf of that file, and in turn reduces the latency of our file system. Whole file caching has been used effectively in several distributed file systems [7, 8, 18]. In a mobile environment the possibility of temporary disconnection and the availability of local storage make whole file caching the best option.

We split the file system traces into eight 24 hour periods lettered A through H. Given the time frame and environment under which these traces were generated, we chose a cache size of four megabytes as a reasonable size for our initial tests. After developing an understanding of how the various parameters effected performance, we explored our model's performance for cache sizes up to 256 megabytes.[3]

3.1 Prefetch Threshold

Our first goal was to examine which probability thresholds would result in the best hit ratios. Figure 2 shows how our hit ratio varied as the threshold settings ranged from a probability of 0.001 to 0.25. From this graph we can see that a setting in the region of 0.05 to 0.1 will offer the best performance. From Griffioen and Appleton's work [6] and our earlier work, we expected this setting to be quite low. Even so, it is surprising that such an aggressive prefetch threshold produced the best results.

To explain this, we first consider that each trace is comprised of over 10,000 distinct files. Since each of these files can be a child to any node, the tree we build will become very wide. Since the count for each parent is the sum of the counts for its children, such a wide tree would result in parents with much higher counts than their individual children. It follows that the parent count divided by the count of children would be rather low even for children that frequently follow their parent.

[3] For readability our graphs only show cache sizes up to 128 megabytes.

For settings greater than 0.025, performance does not change radically with minor variations. However for settings below 0.025 we see a sharp drop in performance as a result of prefetching too many files. Thus we can say that this algorithm is stable for settings of the probability threshold that are greater than 0.025.

3.2 Number of Files Prefetched

We were initially concerned that these low threshold settings might have resulted in prefetching an impractical number of files, but this is not the case. In fact, for a probability threshold of 0.075 the average number of files prefetched per open event ranged from 0.21 to 1.10 files. Figure 3 shows how the average number of prefetches varied for the same settings of probability threshold used in §3.1. This graph shows that for extremely low threshold settings, less than 0.025, the number of files prefetched quickly becomes prohibitive. However, for settings in the region of 0.05–0.1, the average number of files prefetched would not impose an excessive load.

3.3 Maximum Order of the Model

To see how much benefit was gained from each order of modeling, we simulated models of order ranging from zero through four. Since we ignore the predictions of the zero order model, a zero order cache does not prefetch, and is therefore equivalent to an LRU cache. Figure 4 shows how performance varied over changes in model order. While we expected to gain mostly from the first and second orders, the second order improved performance more than we had expected, while fourth and higher orders appeared to offer negligible improvements. We hypothesize that the significant increase from the second order model comes from its ability to detect the combination of some frequently used file (*e.g.* **make** or **xinit**) and a task-specific file (*e.g.* **Makefile** or **.xinitrc**).

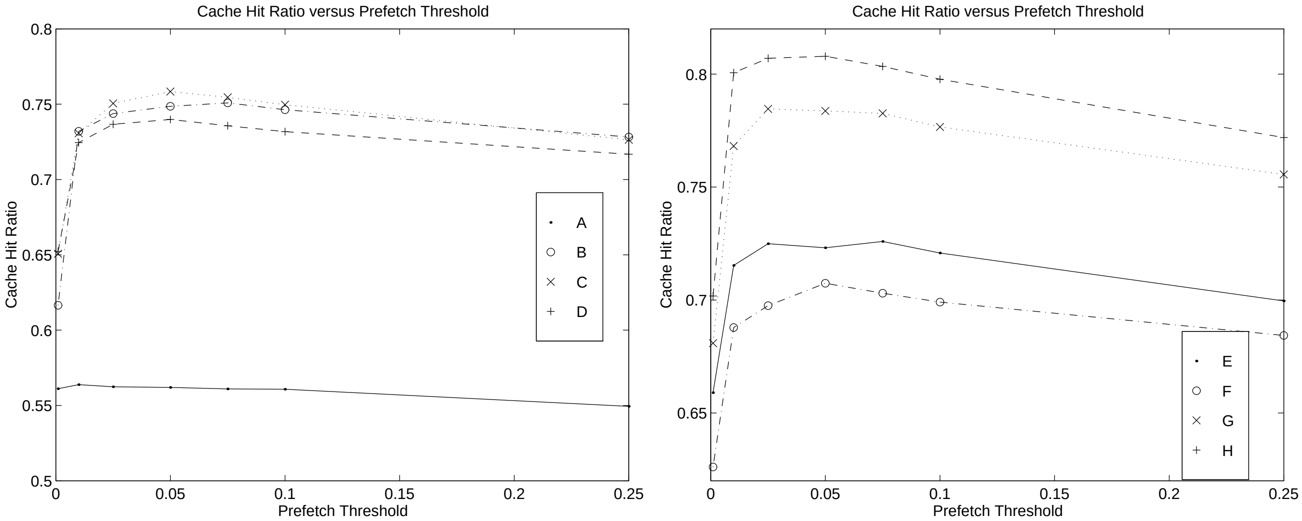


Figure 2: Cache hits versus prefetch threshold (cache size 4 megabytes, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

3.4 Improvements Over LRU

With a firm understanding for the appropriate parameters of our model, we compared our predictive cache to an LRU cache. For this comparison, both caches simulated four megabytes of cache memory. Our predictive cache extended to the second order and prefetched at a conservative threshold of 0.1. Table 1 shows the results of our simulations. Our predictive cache clearly offered significant improvements over the performance of LRU on all eight traces, averaging 15% more cache hits than LRU and, in the case of trace E as much as 22% more.

3.5 Cache Size

One key concern we had was whether the benefit from our predictive cache would quickly diminish as the size of our cache grew. In order to investigate this we simulated an LRU cache and our predictive cache for varying cache sizes up to 256 megabytes. Figures 5 and 6 show how the cache hit ratios varied as we increased the cache size. From these graphs it is clear that tracking file system actions offers a performance gain that will not easily be overcome by increasing the size of an LRU cache. For example, on average it would require a 90 megabyte LRU cache to match the performance of a 4 megabyte predictive cache.

3.6 Model Memory Requirements

The amount of memory required in our current implementation is directly proportional to the number of nodes in our trie. On average a second order model required 238,200 nodes. Since our implementation required 16 bytes per node, the memory required by a third order model should be well under four megabytes. While this model takes almost as much memory as the cache it models, we note that this additional four megabytes seems negligible when compared to the additional 86 megabytes that would be required for an LRU cache to see equivalent performance. It should also be noted that model size is independent of cache size, therefore a 32 megabyte predictive cache would still require less than four megabytes of model space.

Additionally, in our initial implementation we have made no effort to efficiently use memory in our model. We intend to expand on methods used successfully in the compression technique DAFC [13], to limit the

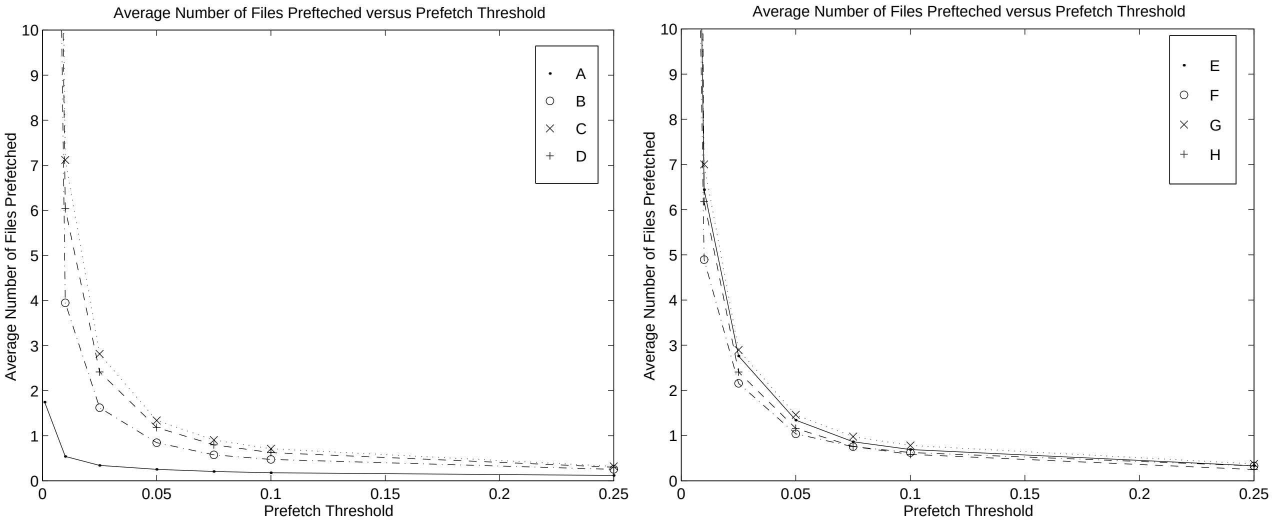


Figure 3: Average number of files prefetched per open event versus prefetch threshold (cache size 4 megabyte, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

Trace	A	B	C	D	E	F	G	H
Predictive	56.1%	74.6%	75.0%	73.2%	77.1%	70.0%	77.7%	79.8%
LRU	48.5%	59.7%	59.8%	57.2%	54.9%	54.0%	58.4%	68.8%
Improvement	7.6%	14.9%	15.2%	16.0%	22.2%	16.0%	19.3%	11.0%

Table 1: Hit ratios for LRU and predictive caches (cache size 4 megabytes, second order model, threshold 0.1).

number of children that each node in our context model has, and to periodically refresh parts of the set of children by releasing links taken up by less frequently seen children. We expect that this modification will not only significantly reduce the memory requirements of our model, but will also allow it to adapt to patterns of local activity. Limiting the number of children a node can have will also ensure that the time required to update our model and predict new accesses is limited to a constant factor.

4 Related Work

Vitter, Krishnan and Curewitz were the first to examine the use of compression modeling techniques to track events and prefetch items [19]. They prove that such techniques converge to an optimal online algorithm. They go on to test this work for memory access patterns [4] in an object oriented database and a CAD system. They deal with the large model size by paging portions of the model to secondary memory, and show that this can be done with negligible effect on performance. Additionally they suggest that such methods could have great success within a variety of other applications such as hyper-text. Our work adapts PPM in a different manner. We avoid the use of vine pointers [2, 10] and instead keep an array of the current contexts. Their method of selection for prefetching (choosing the n most probable items, where n is a parameter of their method) differs from the threshold based method we use. Lastly, the problem domain we examine (file systems access patterns) differs from that which they have worked under (virtual memory access patterns).

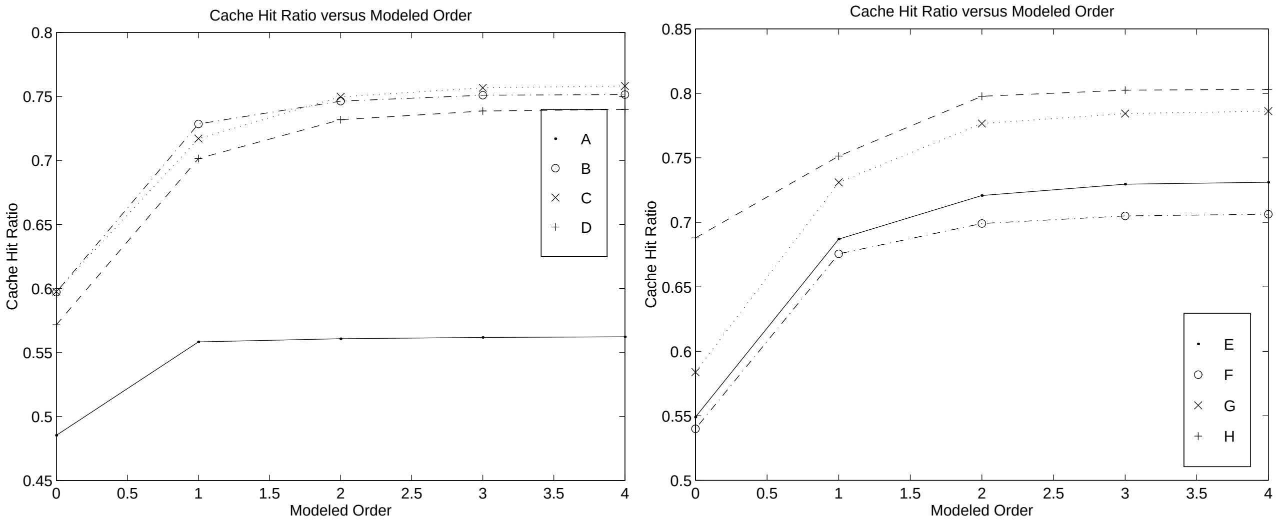


Figure 4: Cache hit ratio versus model order (cache size 4 megabytes, prefetch threshold 0.1).

Within the domain of file systems, Griffioen and Appleton [6] have worked to developed a predictive model that for each file accumulates frequency counts of all files that are accessed in a window after the first. These frequency counts are then used to drive a prefetching cache. Their prediction model differs from ours in that they look at more than just the next event. Additionally, they only consider a first order model. Nevertheless, the method of prefetch selection based on a probability threshold was first presented in their work.

Kuenning, Popek, and Reiher [12] have done extensive work analyzing the behavior of file system requests for various mobile environments with the intent of developing a prefetching system that would predict needed files and cache them locally. Their work has concluded that such a predictive caching system has promise to be effective for a wide variety of environments. Kuenning has extended this work [11], developing the concept of a *Semantic Distance,* and using this to determine groupings of files that should be kept on local disks for mobile computers.

Patterson, *et al.* [16, 15], have modified a compiler and file system to implement a method called *Transparent Informed Prefetching* where applications inform the file system which files to prefetch. While this method can offer significant improvements in throughput, it is dependent on the applications ability to know its future actions. For example **cc** would only know which header files it would need once it had read in the line **#include** ⟨**stdio.h**⟩, while our our predictive model could notice that every accesses to **program.c** causes an access to **stdio.h**. Finally, such an application specific method would not be able to make use of any relationships that exist across applications (*e.g.* between **make** and **cc**).

Cao *et al.* [3] have approached this problem from a unique perspective, by examining what characteristics an off-line prefetching technique would require to be successful.

5 Future Work

The following items are intended as areas of future exploration:

Trie memory requirements – Our current implementation was designed as a proof of concept without concern for the memory usage of our predictive trie. Both Griffioen [5] and Curewitz [4] successfully ad-

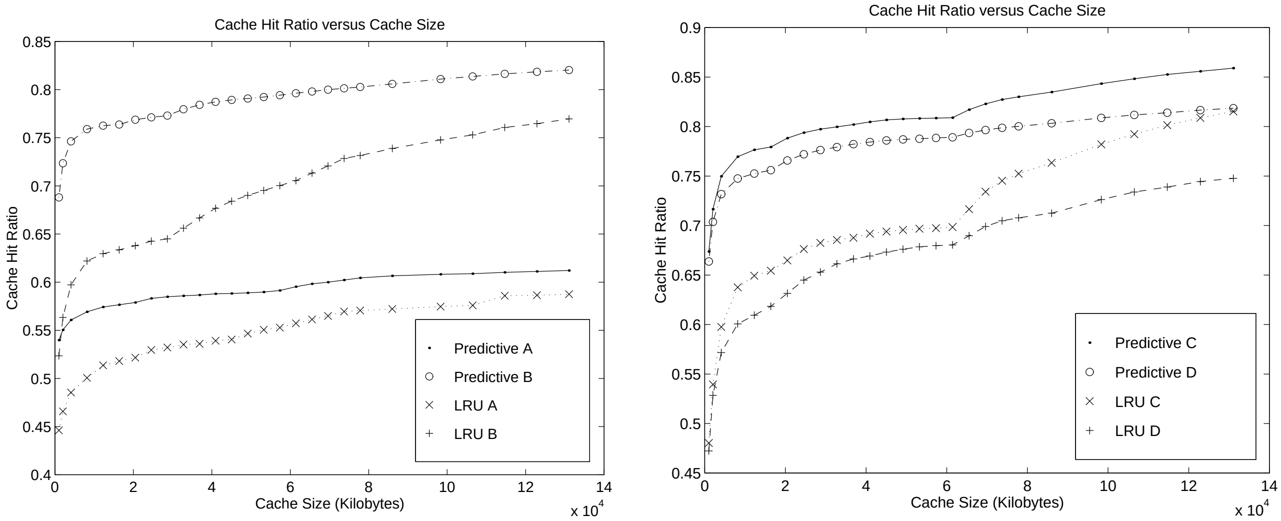


Figure 5: Cache hit ratio versus cache size for both predictive cache and LRU (cache sizes 1–128 megabytes).

dress this issue. We hope to expand on their work, in conjunction with data compression techniques. We expect that our predictive cache will see significantly improved efficiency after we implement a fixed limit on the number of children each node can have. Additionally, methods that give more weight to more recently seen patterns and purge old informations not only reduce the model size, but also can improve performance.

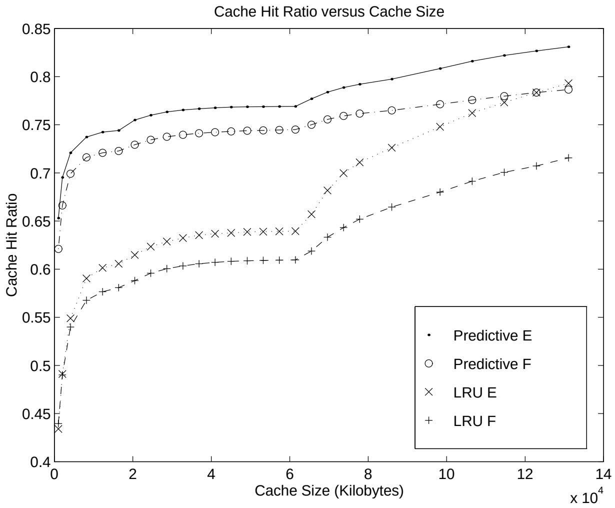
Predicting from grandparents – It is quite possible for one file open event to be the cause of two future events that occur closely. As a result, the order of these two events may vary. For example, an open of file *B* causes opens to files *A* and *C*, so we would have one of two resulting sequences *BAC* or *BCA*. We intend to investigate using all the descendants of a context to predict possible variations in the ordering of events. Using the final tree from Figure 1, if we see an open of file *B*, then we would not only prefetch file *C* but also file *A* as well. Such a forward-looking prediction would enable us to avoid cache misses for the sequence *BAC* as well as *BCA*.

Modifications to the prefetching algorithm – Curewitz's [4] approach to prefetching is to select the n most likely items (where n is a parameter of the model). We intend to investigate a combination of this method and the threshold based selection that we have used by placing a limit on the number of files that can be prefetched. We also intend to investigate the effect of having different threshold settings for each order of the model.

Predictive replacement – While our cache is based on a predictive model to prefetch files, it still uses LRU to determine which files to expel from the cache when space is needed. Using our predictive model to determine cache replacements may offer further improvements in performance.

Read wait times – While cache hit ratios are commonly used to measure the performance of a caching algorithm, we are mostly concerned with the amount of time that is spent waiting for file access events to complete. Our intent is to extend our simulation to include read-wait times allowing us to account for the additional I/O load generated by prefetching.

Selection by cost comparison – Finally we intend to explore a prefetching selection model that uses the estimated probabilities of our model, in conjunction with other factors, such as memory pressure and file size, to estimate the cost, in terms of read-wait times. These estimates would be used in each case to decide if it was more beneficial to prefetch or not. Our hope is



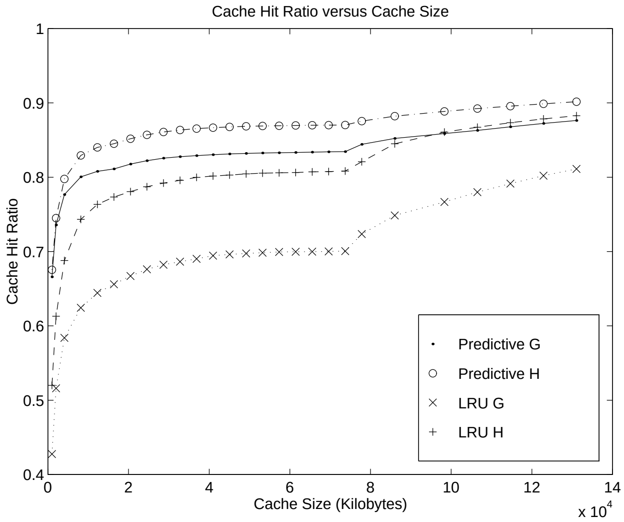


Figure 6: Cache hit ratio versus cache size for both predictive cache and LRU (cache sizes 1–128 megabytes).

that such a parameterless selection method will adjust to changing file system behavior.

6 Conclusions

Our prediction model shows how file system events can successfully be modeled with the multi-order techniques used in PPM. Through trace driven simulations, we have demonstrated that this model can be used effectively to predict future file system events. From our ability to effectively predict future events based on previous file system events, we have shown strong empirical evidence that there exists consistent and exploitable relationships between file accesses.

As the I/O gap widens due to advances in processor design, file system latency will become a greater hindrance to overall performance. By exploiting the highly related nature of file system events we can use methods such as predictive caching to reduce file system latency and improve overall system performance.

7 Acknowledgments

This line of investigation was originally inspired by the comments of Dr. J. Ousterhout regarding relationships between files. We are grateful to Prof. M. Baker for her support in working with the Sprite file system traces, B. Sherrod for his insightful comments and continued support, R. Appleton for sharing advance copies of his work and his experiences, T. Van Vleck for his many comments and his idea of a parameterless selection technique, Dr. L. F. Cabrera for his input and guidance, and to the many other people who offered their time and comments.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. *Proceedings of 13th Symposium on Operating Systems Principles*, pages 198–212. ACM, Oct. 1991.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. *Proceedings of the 1995 SIGMETRICS*. ACM, 1995.

[4] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. *SIGMOD Record*, **22**(2):257–266. ACM, Jun. 1993.

[5] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. *Proceedings of USENIX Summer Technical Conference*, pages 197–207. USENIX, Jun. 1994.

[6] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. *Parallel and Distributed Computing Systems*, pages 165–170. IEEE, Sept. 1995.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *Transactions on Computer Systems*, **6**(1):51–81. ACM, Feb. 1988.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 213–25. ACM, Oct. 1991.

[9] D. E. Knuth. *Sorting and Searching*, volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA, 1973.

[10] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis. Brown University, May 1995.

[11] G. Kuenning. The Design of the Seer Predictive Caching System. *Workshop on Mobile Computing Systems and Applications*, pages 37–43. IEEE, Dec. 1994.

[12] G. Kuenning, G. J. Popek, and P. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. *Proceedings of USENIX Summer Technical Conference*, pages 291–303. USENIX, 1994.

[13] G. G. Langdon and J. J. Rissanen. A Doubly-Adaptive File Compression Algorithm. *IEEE Transactions on Communications*, **COM-31**(11):1253–1255, Nov. 83.

[14] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proceedings of USENIX Summer Technical Conference*, pages 247–56. USENIX, 1990.

[15] H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Transparent Informed Prefecting. *Proceedings of 13th Symposium on Operating Systems Principles*. ACM, Dec. 1995.

[16] H. Patterson, G. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefecting. *Operating Systems Review*, **27**(2):21–34. ACM, Apr. 1993.

[17] K. Trivedi. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice Hall, 1982.

[18] R. van Renesse, A. S. Tanenbaum, and Annita Wilschuts. The Design of a High-Performance File Server. *Proceedings of the 9th International Conference on Distributed Computing System*, pages 22–27. IEEE Computer Society Press., 1989.

[19] J. S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. *Proceedings 32nd Annual Symposium on Foundations of Computer Science*, pages 121–130. IEEE Comput. Soc. Press, Oct. 1991.