UNIVERSITY of CALIFORNIA
SANTA CRUZ

PREDICTING FILE SYSTEM ACTIONS FROM REFERENCE PATTERNS

A thesis submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Thomas M. Kroeger

December 1996

The thesis of Thomas M. Kroeger is approved:

Prof. Darrell D. E. Long, Chair

Prof. Glen G. Langdon, Jr.

Dr. Richard A. Golding

Dean of Graduate Studies

Predicting File System Actions from Reference Patterns

Thomas M. Kroeger

Abstract

Most modern cache systems treat each access as an independent event. Events in a computer system, however are driven by programs. Thus, accesses occur in distinct sequences and are by no means random. The result is that modern caches ignore useful information. In a UNIX file system, when the executable for the program **make** is referenced, it is likely that references to files such as **cc**, **as**, and **ld** will soon follow. To successfully track the file reference patterns, a model must efficiently handle the large number of distinct files, as well as the constant addition and removal of files. To further complicate matters, the permanent nature of a file system requires that this model be able to run continuously, forcing any realistic model to address issues of long-term model buildup and adapting to changing patterns.

We have adapted a multi-order context modeling technique used in the data compression method *Prediction by Partial Match* (PPM) to track file reference patterns. We then modified this context modeling technique to efficiently handle large numbers of distinct files, and to adapt to areas of local activity. From this model, we are able to determine file system accesses that have a high probability of occurring next. By prefetching the data for these events, we have augmented a *Least Recently Used* (LRU) cache with a predictive prefetcher, creating a predictive cache that, in our simulations, averages 14% more cache hits than an LRU without prefetching. We show that on average our four megabyte predictive cache has a higher cache hit ratio than a 90 megabyte LRU cache.

Acknowledgments

I am indebted to the members of my committee for their efforts on my behalf. My advisor, Prof. Darrell Long, has taught me the value of precision, rigor and economy as both a scientist and author. As I became afflicted with repetitive strain injuries and could no longer type, he ensured that I was provided with a voice recognition system, allowing my hands to heal. This thesis has been composed entirely through voice recognition, and would never have happened had he not stepped forward on my behalf. For his guidance, support, patience, care and friendship I am extremely grateful (and I promise never again to redecorate his office, at least not during an election year). Dr. Richard Golding has supported me in this work since its beginning. He has taught me much about the practical workings of both I/O systems and academic systems. Prof. Glen Langdon, has been kind enough to take time from his busy schedule, to support this work during his sabbatical.

Many of my follow graduate students have been invaluable in their technical comments, support and friendship. I am sincerely grateful to Lawrence You, Randal Burns, Theodore Haining, Bruce Sherrod, James Spring, Melissa Cline, David Kulp, and the rest of the Concurrent Systems Laboratory. It is certainly a pleasure to consider these people my colleagues.

This line of investigation was originally inspired by the comments of Dr. John Ousterhout regarding relationships between files. Prof. Mary Baker provided support in working with the Sprite file system traces. Randy Appleton and Geoff Kuenning shared their experiences and comments. Dr. L. F. Cabrera gave input and guidance. I am grateful to these and the many other people who offered their time and support.

I am grateful to Dr. Jeffrey Mogul for his support and guidance during my summer at Digital's Western Research Laboratory. Dr. Peter Honeyman is responsible for inspiring me to take an interest in research. The Office of Naval Research has funded my work as a graduate research assistant.

Finally, I must acknowledge my family: my sister Alexandra Thomas who will hold my hand through any surgery and my mother Hildegard Kroeger who forced me to learn how to read so I wouldn't get "stuck in the third grade." Their support and love has been my foundation.

This thesis is dedicated to the memory of Joseph Mund.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While the past few years have brought significant gains in processor technology, permanent storage devices have at best seen moderate increases in speed. The resulting bottleneck from I/O system latency has inspired several researchers to re-evaluate how caching is done [9, 20, 24, 6, 28, 13, 16].

The purpose of a cache is to reduce latency by using limited high speed storage to buffer data between a *workload* and a *data resource* (Fig. 1.1). The workload provides two streams: I/O system requests and write data. It will expect request results and read data. The data resource accepts I/O system requests and write data, while providing requests results and read data. The cache fits in between the workload and the data resource, managing a pool of relatively faster storage to reduce the latency seen by the workload. To manage these resources a cache system will make decisions about *replacement*, *prefetching* and *writing*. The policies by which these decisions are made constitute the algorithm used for caching. This thesis presents a method to track patterns within the request stream and generate a probabilistic model for future requests. This model can be used to help make these decisions. We show how a prefetching policy might use this information.

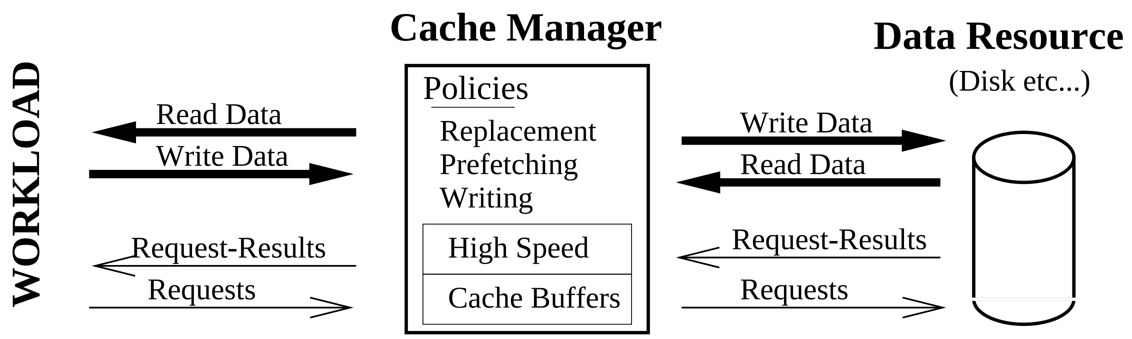

Figure 1.1: Cache System Model.

In an effort to improve I/O performance several researchers have taken the approach of adapting the workload to provide information about *future requests* [13, 24, 6]. With this *informed model* they present several methods for managing cache replacement in conjunction with prefetching [13]. Since write behind policies can mask update latency, their efforts focus on evaluating prefetching and replacement together.

With the same goal of reducing I/O latency several other researchers have pursued methods to determine future requests based on previous workload request patterns [9, 20, 28, 16]. While these *predictive methods* vary in their approach, they are all based on the same idea: that previous action patterns provide indications for future references.

Informed and predictive caching techniques differ in the following ways. Work with informed methods has focused on the problem of how to best utilize caching resources given near-perfect knowledge of future requests, while the work with predictive methods focuses on how we can gain probabilistic knowledge of these future requests from previous actions. Informed methods require modification of the workload itself, making the program designer's or compiler's task more complex, while predictive methods work without modifying the workload. Finally, the class of I/O actions improved by informed prefetching intersects the class of actions predictive prefetching improves, but both methods also have unique situations where they will gain when the other cannot. In other words, there exist situations where informed prefetching will benefit and predictive prefetching will be ineffective, situations where the reverse is true, and situations where both methods will prove effective. The bottom line is that these techniques are in no way mutually exclusive, and a cache that combines both of these techniques should be able to get the best of both worlds.

Predictive prefetching methods are based on the hypothesis that previous reference patterns offer information about future references. The motivation for this hypothesis is as follows. In a computer system, programs define the sequence of actions, and while the actions of a program may vary widely from one invocation to the next, in every case this program has a fixed set of instructions that it must follow. In the common case, the program executes the same sequence of instructions causing a similar set of I/O references and little variation. Take, for example, the case of **pine**, a commonly used mail reader. Upon start-up this program will look for a system-wide configuration file (*e.g.* **/usr/local/pinerc**) and take actions based on that file. For most systems, this file rarely changes and actions based on this file will repeat every time a user invokes **pine**. Next, the program will attempt to find a user specific configuration file and take actions based on it. Again, this file rarely changes. From this file, **pine** may take actions specific to that user, such as reading the user's mail aliases and signature file. Finally, **pine** will sequentially read the user's mail file. As another example, consider the program building process. When a user executes the program **make**, it will often result in references to the files **cc**, **as**, and **ld**. If we note a reference to the file **make** and a specific **Makefile** then another sequence of references:

program.c, **program.h**, **stdio.h**, ... , is likely. These two examples illustrate how a file reference patterns can provide information about future references. Thus, a file system buffer cache that tracks file reference patterns and notes predictive sequences should be able to exploit the information in such sequences, by determining likely references and prefetching that data before it is requested.

We have developed such a predictive prefetching cache [16]. This cache is composed of two parts: a *model* that tracks the file reference patterns and a *selector* that uses this information to prefetch data that is likely to be needed. Our model tracks previous file references patterns through a finite multi-order context modeling technique adapted from the data compression technique *Prediction by Partial Match* (PPM) [4]. We modified this model to efficiently handle a large number of distinct files and adapt to changing reference patterns. Our selector examines the most recently seen patterns and the counts of the events that have followed them to determine which events are likely.

Using these predictions, we augment an LRU cache by prefetching data that is likely to be accessed. The result is a predictive prefetching caching algorithm that, in our simulations, averaged hit ratios better than a non-prefetching LRU cache more than 20 times its size.

The rest of this thesis is organized as follows. Chapter 2 details how we adapted PPM to model reference patterns and used to drive the prefetching. Chapter 3 presents our initial simulations and results. Chapter 4 details how we modified context modeling to work within efficient space constraints and to quickly adapt to changing reference patterns. Chapters 5 and 6 discuss related and future work respectively. Chapter 7 concludes this thesis.

Chapter 2

Predictive Prefetching Method

This chapter presents the details of how we adapted data compression modeling to model the request stream, and how this information was used to implement a predictive prefetching cache. Using previous request patterns this cache prefetches probable future accesses and hence improves I/O performance.

As in data compression, where a model drives a coder, our predictive cache can be divided into two portions: a *model* that drives a *selector*. The model tracks reference patterns and produces probability distributions for future events. The selector uses these distributions to select which files to prefetch. Figure 2.1 illustrates how these components integrate into a I/O cache.

Our model tracks observed file reference patterns through a finite multi-order context modeling technique adapted from *Prediction by Partial Match* (PPM) [4]. This model uses a trie [15] to store file reference patterns and the number of times they have occurred.

To determine likely future references our selector compares the counts of the most

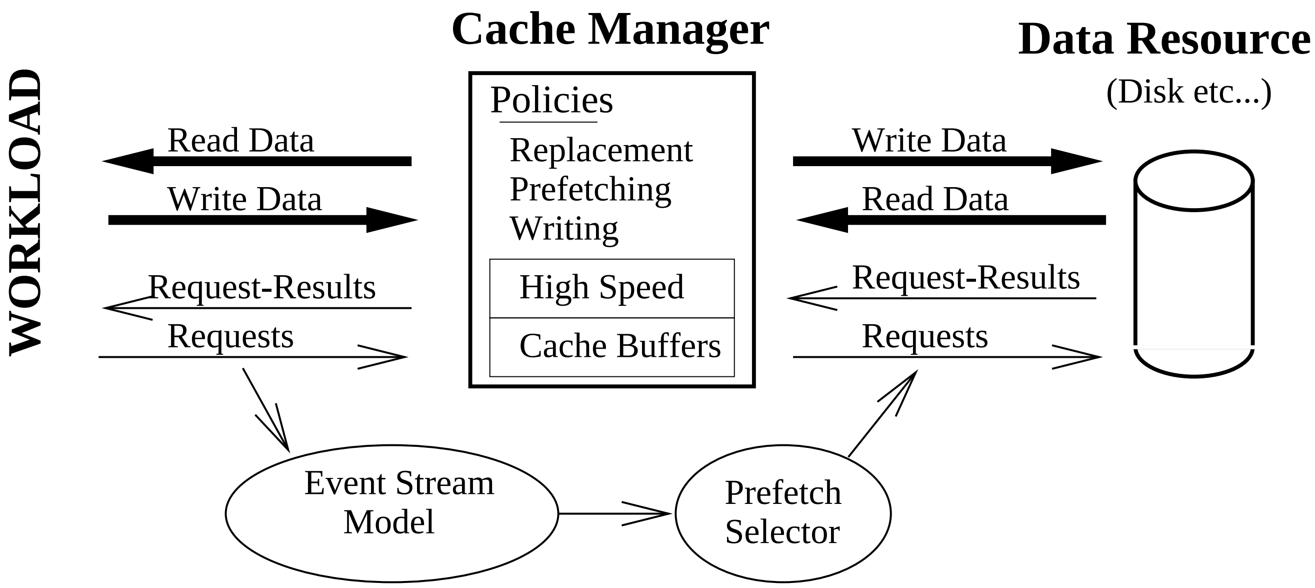

Figure 2.1: Cache System with Predictive Prefetching.

recently seen patterns (the *current contexts*) to the counts of the reference events that have previously followed these contexts. Using these predictions, we augment an LRU cache by prefetching data that are likely to be referenced.

In the next section, we will provide a short background of context modeling from the field of data compression. In §2.2 we explain how we adapted these context modeling techniques to track file reference patterns, §2.3 explains how the selector uses the information tracked to improve cache performance, and §2.4 analyzes the limitations in adapting context modeling to file systems.

2.1 Context Modeling

Modeling previously seen event patterns to predict upcoming events is a problem faced in many domains. In this work we intend to track file access events to predict future accesses. In text compression we model characters in a text file to determine the probability distribution for the next character. In either case we can consider the events modeled as symbols be they file accesses or characters in a text file.

In data compression, context modeling is a common technique that counts the frequency of symbol patterns seen. By comparing the counts of patterns currently seen with the counts the symbols that have previously followed these patterns, such models provide a distribution of likelihood for the next symbol. A trie, a data structure based on a tree, is often used to efficiently store previously seen patterns and to maintain counts of each pattern's frequency.

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**". In text compression a context is used to model which characters are likely to be seen next, and thus determine which symbols should be encoded with the least number of bits. 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**". Data is then compressed by encoding "**i**" with fewer bits at the cost of having to encode an "**h**" with more bits.

The length of a context is termed its *order*. In the example string, "**jec**" would be considered a third order context for "**t**". In fact we could describe the next character as occurring under any of the following contexts, "**t**," "**ct**" "**ect**," "**ject**," "**bject**" and "**object**." Techniques that track multiple contexts of varying orders are termed *Multi-Order Context Models* [4]. 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 a string, its current state 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 or *current context* of what we are modeling, be it a character input stream or an I/O system reference pattern. With each new event or character, the set of new contexts that will model the next state is generated by appending the newly seen item to the end of the contexts that model the previous state. If {ø, "**c**," "**ec**," "**jec**"} is our current context at time t, and at time $t + 1$ we see the character "**t**," our new context 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 [15]. A trie is a data structure based on a tree that is used to efficiently store sequences of symbols (*e.g.* storing sequences of letters from the alphabet). Each node in this trie contains a symbol (*e.g.* a letter from the alphabet). By listing the symbols contained on the path from the root to any individual node, we use each node to represent a sequence of symbols (*e.g.* previously seen contents). The result is that the children of every node, represent all the symbols that have been seen to follow the sequence represented by the parent.

To do context modeling, we add to each node a count of the number of times that pattern (or context) has been seen. Since the children of a node represent all of the symbols that have previously followed that nodes's sequence, then the sum of their counts should equal the count of that node. The one exception to this case is when the node represents a current context, in which case we have just seen this sequence and have not yet seen a successor. In this case, the frequency count is equal to the sum of its children's counts plus one. Therefore, we can use the formula $count_{child}/(count_{parent} - 1)$ to estimate the likelihood of a child's symbol occurring as the next event.

2.2 Tracking File Reference Patterns

A predictive prefetching policy needs a model that tracks file references to predict which accesses are likely to occur next. For this purpose we have adapted finite multi-order context models to model file reference patterns. By replacing letters from the alphabet with unique file identifiers (*e.g.* inode numbers) each node in the trie contains a specific file identifier. Through its path from the root, each node represents a sequence of file references that have been seen. These file reference patterns are the contexts that we use to indicate the state of our file system. As before in each node we keep a count of the number of times this pattern has occured.

To easily update this trie and use it to determine likely future events, we maintain an array of pointers, indexed by context order, 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 pattern (the new C_{k+1}) has occurred before, and is represented by this node's child, so we set the new C_{k+1} to point to this child and increment its count. If no such child is found, then this is the first time that this pattern has occurred, so we create a child denoting the event A and set the $k + 1^{\text{st}}$ element of our array to point to this node. Once we have updated each context in our set of current contexts, we have a new state that describes our file system. Figure 2.2 extends an example from Bell [4] to illustrate how this trie would develop when given the sequence of 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.

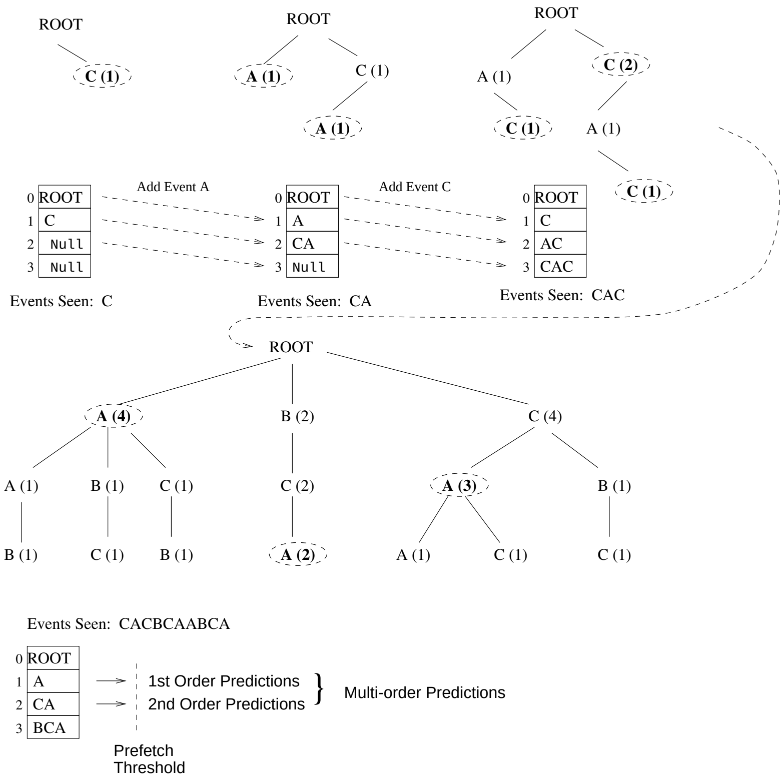

Figure 2.2: Example tries for the sequence $CACBCAABCA$.

2.3 Selecting Events to Prefetch

Given the probability information maintained by the model, we present one possible method of using this information to improve cache performance, by prefetching data that is likely to be needed. To select which data to prefetch we compare the likelihood estimate for each child of a current context to a prefetch threshold, which is a parameter of the model.

To determine which files should be prefetched, after each reference we compare the counts of each new current context with the counts of their children. Using the formula from §2.1 we estimate the likelihood of a reference to each child's file. If the estimated likelihood is greater than or equal to the prefetched threshold parameter, 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 predicts based on frequency counts of each file and thus, would always cause prefetching of the most frequently accessed files, making the predictive cache become an LFU cache. Therefore, we have chosen to ignore the zero order predictions for this study. The result is that since a zero order predictive model will do no prefetching, a zero order predictive cache does not augment the the original LRU cache.

Additionally, for this study, we chose to treat prefetch data as if it had been referenced by placing it at the front of our cache. Since cache replacement is still LRU, the data is likely to remain in the cache for the next several events. So long as the event occurs before its data is removed from the cache we avoid a cache miss.

2.4 Limitations of Context Modeling

Adapting the PPM model to a file system reference stream brought out two important differences between modeling text and file system reference streams. These differences, the number of symbols modeled and the length of stream, serve to emphasize the limitations of model space requirements and adapting to changes in reference patterns.

The set of symbols modeled, also called the *alphabet*, for a file system is larger and more dynamic than the alphabet for a text file. The alphabet size of a text file is simply the number of characters in the encoding (*e.g.* 256 characters in ASCII) used for the text file. The number of files in a common file system is several orders of magnitude larger, and the alphabet itself is constantly changing as files are created and deleted.

The second difference is that the lifespan of a file system, or the length of the file reference stream, cannot be predetermined. Therefore, an effective model must be able to continue indefinitely. While a text file is normally of a set length, having a known beginning and end, the stream of references in a file system can continue over several years of operation, and the end of this stream is not known until it arrives.

The first problem these differences cause is that of model space requirements. Since each node in the trie can have a child for every member of the alphabet, the trie built under the PPM model does not scale well to the increased alphabet size of a file system. In fact, the model size is $O(a^m)$ where a is the alphabet size, and m is the maximum order modeled. Additionally, since at each event we must scan all the children of each node in the current context, the computational complexity of this model is $O(am)$. While in our simulations the model size was significantly less than $O(a^m)$, the unbounded stream length points out that this model must be able to live indefinitely, and over greater lengths of time the space requirements would grow toward this bound.

The unbounded length of a file system reference stream will also cause large counts to build up over extended periods of modeling, making it difficult for a model to learn new predictive patterns. For example, when a programmer modifies a specific **Makefile** the references that follow an access to this **Makefile** can also change. Under the context model descried in this Chapter, if this **Makefile** has seen 100 references then (given a prefetch threshold of 0.1)[1] the new patterns from the modified **Makefile** would need to occur at least 11 times before they would be used. Also, data from the old reference patterns could be incorrectly prefetched for an additional 900 references. Therefore, for a model to be effective over extended durations it must adapt to changes in the each file's predictive nature.

In Chapter 4 we show how we modify context modeling to address these issues. Nevertheless, before we deal with the issues of model size and adapting, we must first demonstrate that context modeling is effective in predicting file references. To this end, Chapter 3 details the results from our trace driven simulations of predictive caching based on context modeling.

[1]Chapter 3 presents 0.1 as an effective setting for prefetch threshold

Chapter 3

Simulations and Results

In order to test our hypothesis that reference patterns offer useful information about future events, we simulated the performance of a normal LRU cache without prefetching and the performance of a cache with predictive prefetching. The primary purpose of our experiments was to investigate the effectiveness of using previous reference patterns to drive cache prefetching. We used cache hit ratios as a measure of this effectiveness. As a secondary objective we used these simulations to explore the additional load that would result from this prefetching. Finally, we examined what settings of model parameters would produce the best hit ratio and how sensitive our model is to variations in these parameters.

We found that using reference patterns to drive prefetching for a four megabyte cache, resulted in a 15% higher hit ratio. Moreover, for an LRU cache to see the same hit ratio, it would need 90 megabytes of RAM. Using these simulations to investigate model parameters we found that, for the trace data used, the prefetch threshold is best set in the range of 0.05–0.1, model order was best set at 2, and model performance was stable for any reasonable variations in these parameters.

3.1 Experimental Methodology

We used discrete event simulations to model the cache hit ratios of a normal LRU cache and a LRU cache with predictive prefetching (called a predictive cache). Using file open events from the Sprite file system traces [3] to generate a workload, we measured the hit ratios of both caches and the number of files prefetched by a predictive cache, across variations in cache size, prefetch threshold and model order.

We developed a discrete event simulator that modeled the actions of an LRU cache, using whole file caching. We chose to consider whole file caching for three reasons. First, 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. Secondly, whole file caching has been used effectively in several distributed file systems [12, 14, 26]. Finally, in a mobile environment the possibility of temporary disconnection and the availability of local storage make whole file caching desirable. The cache itself was divided into one kilobyte blocks.

We augmented this LRU simulation to model reference patterns and prefetch data as described in Chapter 2. Our primary objective was to determine if reference patterns can offer information that indicates future references, so we chose to ignore the issues of scheduling prefetching events and to model prefetching as an instantaneous event.

To simulate the workload of a file system we used file open events from the Sprite file system traces [3]. These traces represent the workload of a distributed file system in an academic research environment. We split these traces into eight 24-hour periods called A through H, to provide eight separate reference streams. Table 3.1 shows the number of unique files and the total number of open events in each trace.

We used cache hit ratio as a metric for initial comparison and to offer some insight into the latency reduction. To gain a more tangible appreciation for the significance of our increased hit ratio, we then determined how much additional RAM an LRU cache would need to achieve the same performance increase. To measure the additional work load incurred from predictive prefetching we measured the number of files prefetched after each reference. Using these metrics for comparison, we explored how variations in prefetch threshold, model order and cache size effected cache performance.

3.1.1 Bounds on Performance

In order to better evaluate the improvement seen with predictive prefetching, we present the following bound on the performance of an online predictive prefetching cache $h \leq 1 - (f/e)$, where h is the hit ratio of an online predictive prefetching cache, f is the number of unique files referenced, and e is the number of events seen in the trace. The basis for this bound is as follows. Any online predictive prefetching cache must see at least one reference to a file before it can predict future references to that file. Therefore, we bound our model by knowing that it must miss at least once for each unique file referenced in the trace stream. The last row of table 3.1 shows this bound.

Table 3.1: Counts of files and events for each trace and a bound on the hit ratio of a online predictive prefetching cache.

Trace	A	B	C	D	E	F	G	H	Ave.
Files	16310	45211	19376	24250	23259	40573	13061	21098	25392.3
Events	42923	288068	202482	175515	181972	257670	171017	325848	205686.8
Hit Ratio Bound	62.0%	84.3%	90.4%	86.2%	87.2%	84.3%	92.4%	93.5%	85.0%

3.2 Improvements Over LRU

Table 3.2 provides a direct comparison of both caches simulated at a size of four megabytes. Our predictive cache prefetched at a threshold of 0.1 and modeled up to second order events. These parameters were selected based on empirical tests that are detailed in §3.3.

Table 3.3 shows what fraction of the bound each cache was able to achieve. From these tables one can see that our predictive cache clearly offered significant improvements over the performance of LRU on all eight traces, averaging 15% more cache hits than an LRU and achieving more than half of the possible improvement.

Table 3.2: Hit ratios for LRU and Predictive caches (cache size four megabytes, second order model, threshold 0.1).

Trace	A	B	C	D	E	F	G	H	Ave.
Bound	62.0%	84.3%	90.4%	86.2%	87.2%	84.3%	92.4%	93.5%	85.0%
Predictive	56.1%	74.6%	75.0%	73.2%	72.1%	70.0%	77.7%	79.8%	72.3%
LRU	48.5%	59.7%	59.8%	57.2%	54.9%	54.0%	58.4%	68.8%	57.7%
Increase	7.6%	14.9%	15.2%	16.0%	17.2%	16.0%	19.3%	11.0%	14.6%

Table 3.3: Fractions of Bound Achieved for LRU and Predictive caches (cache size four megabytes, second order model, threshold 0.1).

Trace	A	B	C	D	E	F	G	H	Ave.
Predictive	0.905	0.885	0.830	0.849	0.827	0.830	0.841	0.853	0.851
LRU	0.782	0.708	0.662	0.664	0.630	0.641	0.632	0.736	0.679

To investigate whether the benefit from our predictive cache would quickly diminish as the size of our cache grew, we simulated an LRU cache and our predictive cache for varying cache sizes up to 120 megabytes. Figure 3.1 shows how the average cache hit ratios varied as we increased the cache size. This graph show that the performance gains from prefetching based on previous reference patterns will not easily be overcome by increasing the size of an LRU cache. For example, on average it would require an LRU cache with more than 90 megabytes of memory to match the performance of a 4 megabyte predictive cache. It should also be noted that the bound presented in §3.1 is also a bound on the hit ratio of an LRU cache as cache size increases. Therefore, the hit ratios for these two algorithms will eventually converge as cache size is increased. However, from Figures 3.1 and 3.2 we can see that for any practical cache size a predictive cache will be much closer to this bound.

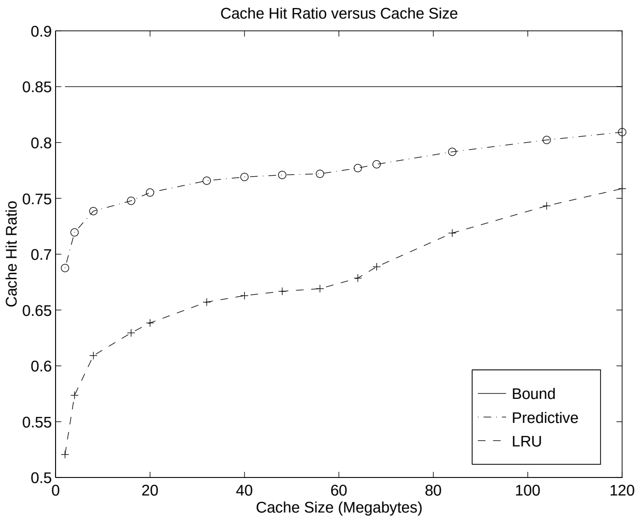

Figure 3.1: Average cache hit ratio versus cache size for both predictive cache and LRU (cache sizes 1–120 megabytes). Figures 3.2 and 3.3 provided details by trace.

3.2.1 Number of Files Prefetched

To determine of the additional load placed on a file system, we measured the average number of files prefetched per reference event. Figure 3.4 shows how the average number of prefetches varied for settings of the probability threshold ranging from 0.001 to 0.25. This graph shows that for extremely low threshold settings, less than 0.025, the number of files prefetched quickly becomes prohibitive. However, for settings of 0.05 or

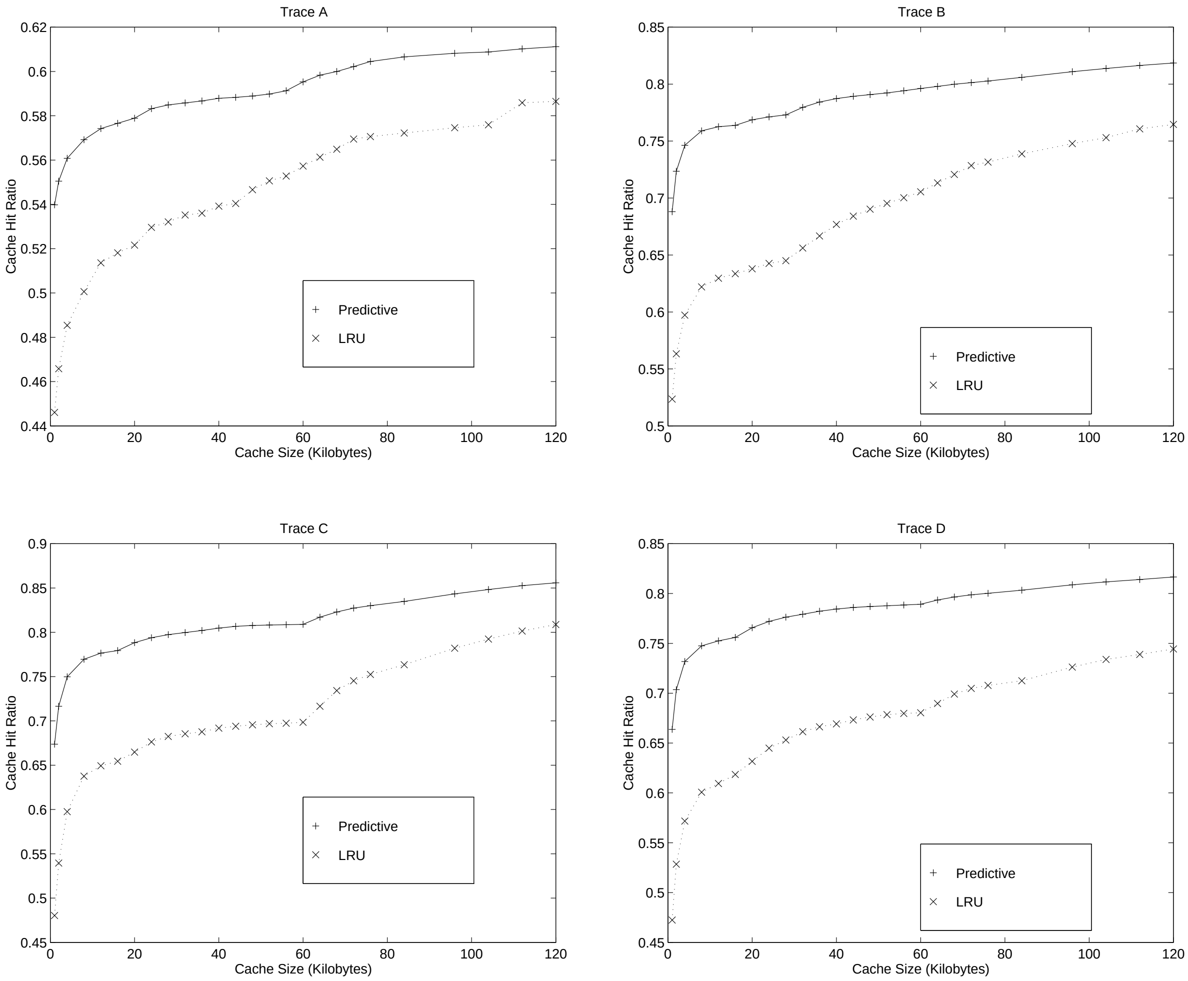

Figure 3.2: Cache hit ratio versus cache size for traces A through D (2–120 megabytes, second order predictive models, prefetch threshold 0.1).

greater, the average number of files prefetched would not impose an excessive load. 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.

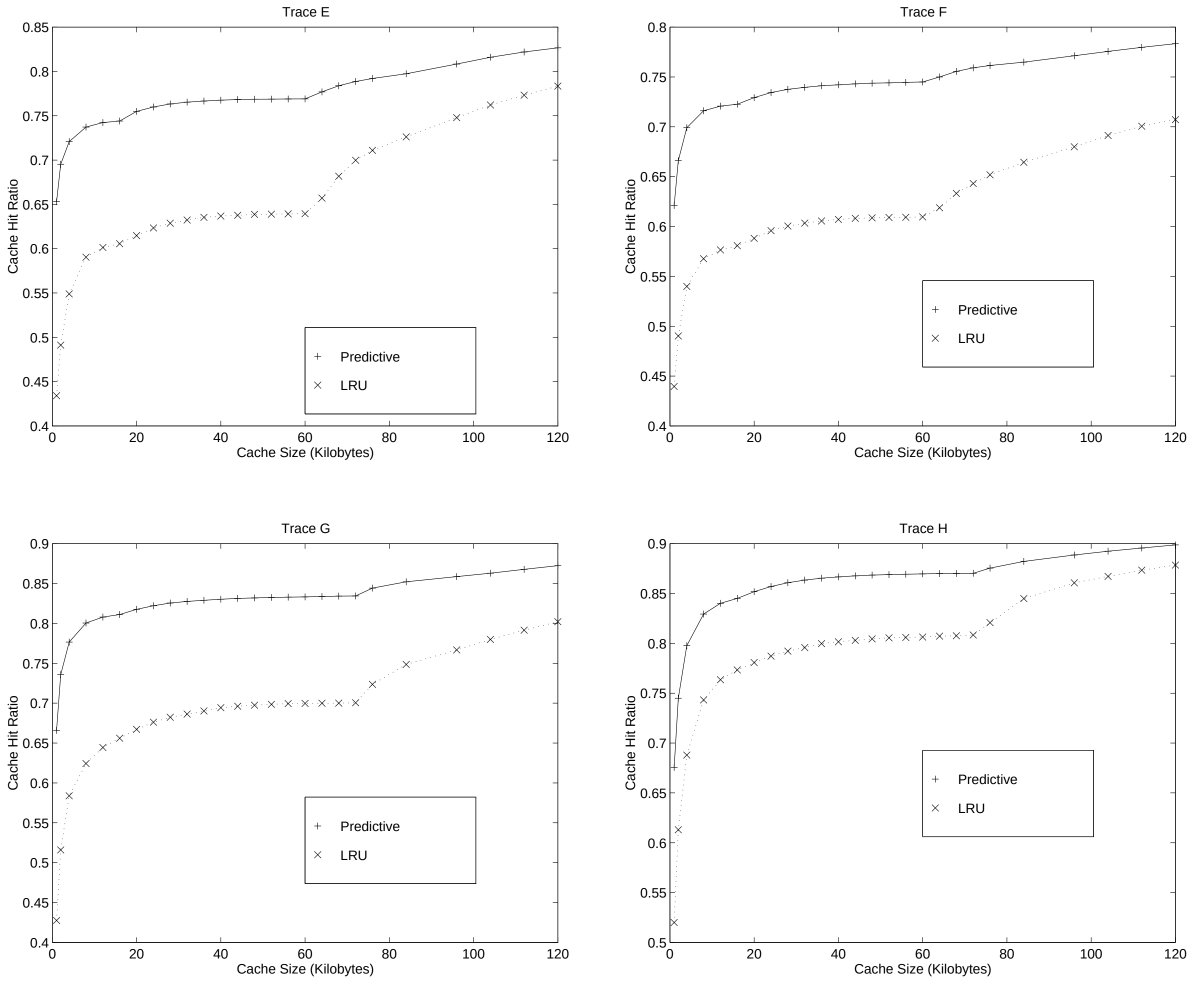

Figure 3.3: Cache hit ratio versus cache size for traces E through H (2–120 megabytes, second order predictive models, prefetch threshold 0.1).

3.3 Parameter Settings

In this section we discuss how the parameters of prefetch threshold and model order affected predictive cache performance. This investigation had two goals. First, we intended to determine what values of each parameter would offer the best performance. Second, we wanted to examine the sensitivity of our model with respect to variations in these parameters. A model whose performance changes significantly with small variations

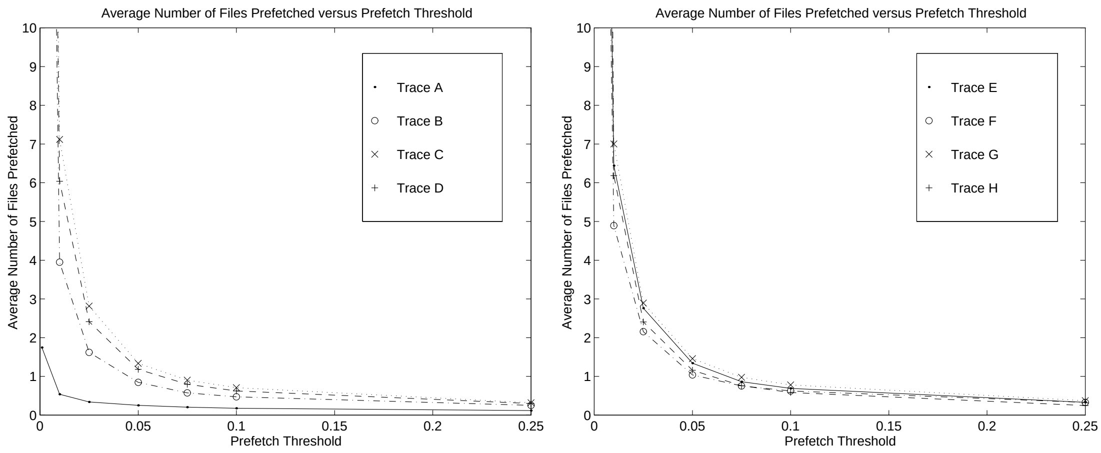

Figure 3.4: Average number of files prefetched per open event versus prefetch threshold (cache size 4 megabyte, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

in parameters is called parametrically instable. This stability is important because, if slight variations in a parameter would cause significant changes in performance, then it is quite possible that for slightly different workloads the best setting for these traces would have significantly different results. In all cases the parameters appeared to be stable within reasonable limits.

3.3.1 Prefetch Threshold

We found a prefetch threshold in the region of 0.05 to 0.1 offers the best hit ratios. Figure 3.5 shows how our hit ratio varied as the threshold settings ranged from a probability of 0.001 to 0.25. From Griffioen and Appleton's work [9], we expected this setting to be quite low. Even so, it is surprising that such an aggressive prefetch threshold produced the best results. The fact that cache hit ratios increases as the threshold setting decreases to 0.1 indicates that on average a file with a likelihood as small as 0.1 is more useful in the cache than the data it would replace.

With regards to stability, 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 the result of prefetching too many files, which pushes needed data out of the cache. Thus, for the workload represented by these traces, we can say that this algorithm is stable for settings of the probability threshold that are greater than 0.025.

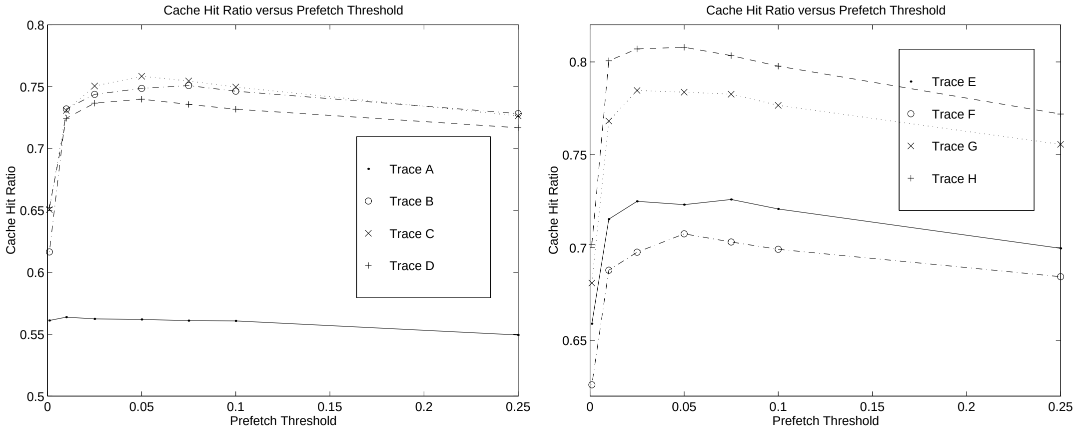

Figure 3.5: Cache hits versus prefetch threshold (cache size four megabytes, second order, threshold settings 0.001, 0.01, 0.025, 0.05, 0.075, 0.1 and 0.25).

3.3.2 Maximum Order of the Model

To determine the benefit 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 thus equivalent to an LRU cache. Figure 3.6 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**).

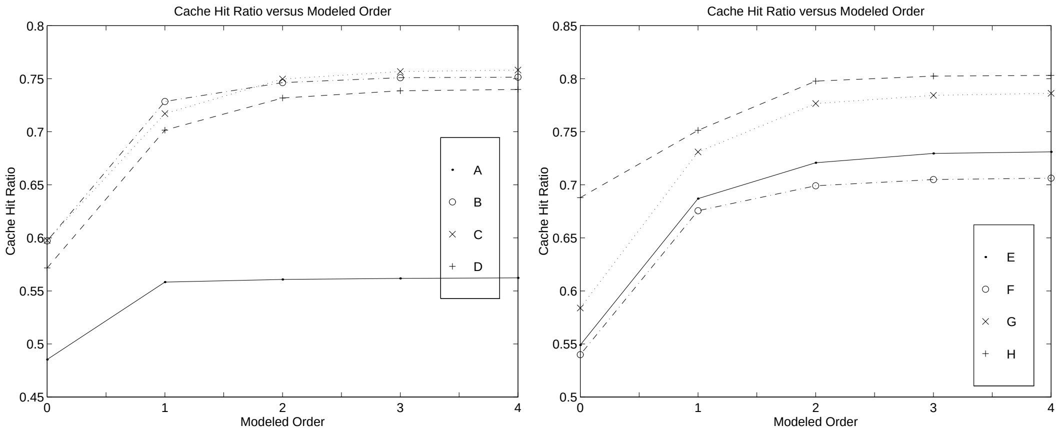

Figure 3.6: Cache hit ratio versus model order (cache size four megabytes, prefetch threshold 0.1).

3.4 Interpretation of Results

From these simulations we have seen that prefetching based on reference patterns can provide a significant increase in cache hit ratio – an increase that, for a four megabyte LRU cache, is equivalent to the addition of 86 megabytes of RAM. By measuring the number of files prefetched we have seen that the load imposed by this prefetching is reasonable. Finally, since performance did not change radically with small variations in the prefetch threshold and model order, we can conclude that this model is stable for the workloads presented in the Sprite traces.

Chapter 4

Partitioned Context Modeling

In adapting context modeling to track file references, the differences in the nature of the alphabet and the length of the symbol stream brought out the importance of model space requirements and the ability to adapt to changes in predictive patterns. This chapter details how we have addressed these issues by modifying the context modeling technique presented in Chapter 2. This new technique, *Partitioned Context Modeling*, partitions the trie based on first order nodes and then statically limits the size of each partition. We adapted our predictive cache simulator to implement this new model and saw a minor loss in cache hit ratio and a significant reduction in model space requirements.

The amount of memory required by the context model is directly proportional to the number of nodes in the trie. The traces evaluated in Chapter 3 generate 238,200 nodes, on average. Since this implementation required 16 bytes per node, the memory required by a second order model should be less than 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 a cache without prefetching to see equivalent performance. Nevertheless, this only represents the model buildup that occurred over a 24 hour time span. Over longer periods of time this model would increase in size.

Other predictive caching methods used virtual memory to page parts of this trie to secondary storage, and have shown that this can be done without losing all performance gains [28, 20]. However, efficiently restricting the buildup of model size has not been addressed within the domain of predictive caching. Within the data compression community efficiently restricting the growth of the PPM model has been addressed in a limited manner [21].

4.1 Improving Context Modeling with Partitions

In order to retain the predictive nature gained from each file we have pursued a model that retains all first order nodes and reduces space requirements by limiting the number of descendants of each first order node. The result is that model space requirements are reduced from $O(a^m)$ to $O(a)$. The model RAM requirements are then further reduced by paging each partition with the object it represents. The result is that the RAM requirements for a predictive cache become directly proportional to the cache size, while requiring $O(a)$ on disk.

Examining the trie built by the model in Chapter 2, we observe that the number of first order nodes corresponds to the number of distinct files that have been referenced. In fact, each first order node represents a reference pattern of length one, consisting of one reference to the file represented by that node. Additionally, note that each first order node and its descendants represent all previously seen reference patterns beginning with a reference to that file. Since removing any first order node would lose all the information associated with patterns that begin with the file represented by that node, we have pursued an approach to limit the model size by not purging any first order nodes but instead limiting their descendants.

This approach divides the trie into partitions, where each partition consists of a first order node and all of its descendants. The number of nodes in each partition is limited to a static number that is a parameter of the model. The effect of these changes is to limit the total model size to $O(a)$, and the computational complexity to a constant. Figure 4.1 shows a modified trie with these static partitions.

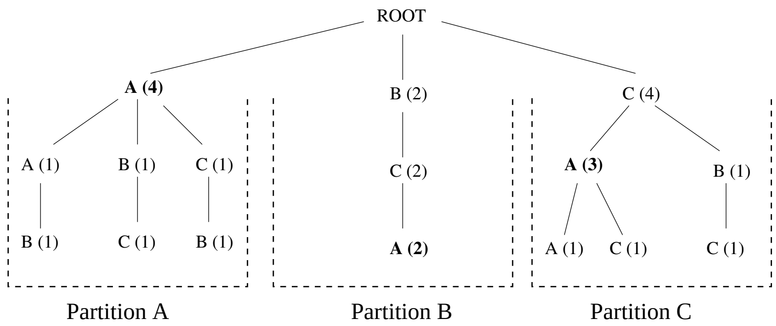

Figure 4.1: Example partitioned trie for the reference sequence $CACBCAABCA$.

Fixed-size partitions require modifying how references are handled. As each reference occurs counts are updated as before. However, when a new node is needed in a partition that is full, the model must either clear space or ignore this reference. Currently, when this situation occurs all node counts in the partition are divided by two (integer division), then any nodes with a count of zero are cleared to make space for new nodes. If no space becomes available, the reference is ignored. Additionally, when new reference patterns occur they will cause node counts to be reduced, resulting in a bias towards more recent references, thus adapting to new reference patterns.

Another result of partitioning the trie in this manner is that the only partitions actively being modified are those whose first order node represents an object currently in the cache. As a result, the RAM memory requirements of this model can be reduced even further by paging the partition with the object that it represents, thus splitting the trie into two parts: the inactive paged partitions and an *active trie* containing the partitions of those files currently in the cache. For example, in a Unix file system one could make the partition part of an extended i-node structure. Then as each file is fetched from the disk and cached in the disk buffer cache its partition would also be read in and held in an active trie. This active trie would hold the partitions for each file in the cache, allowing the predictive cache to update these partitions as references occurred. As each file is flushed from the cache its partition would also be flushed. The result, is that the model memory requirements become directly proportional to number of blocks in the cache.

4.2 Simulating a Partitioned Model

To explore the performance of this partitioned model, we modified our predictive cache simulator to implement it. In this experiment our primary goal was to see how closely the partitioned model approximates the behavior of a non-partitioned model. Our secondary objective was to examine how variations in partition size effect model performance, in order to determine the best setting and the model stability. We again used the Sprite traces to simulate predictive cache performance for the same ranges of cache size used in Chapter 3 and partition sizes ranging from two to 64. We found that with a partition size of 16 our partitioned model would closely approximate the behavior of an unpartitioned model while using far less memory.

Table 4.1 shows the hit ratio of the three caching algorithms, and how the partitioned predictive cache compares to the other two methods. Figure 4.2 shows the average cache hit ratio versus cache size for the three caching methods. Figures 4.5 and 4.4 show the same comparison for each individual trace. Both predictive caches use a second order model and a prefetch threshold of 0.1. Partition size is limited to 16 nodes. These graphs and tables show how a partitioned model closely approximates the behavior of a full trie, averaging a difference of less than one percent.

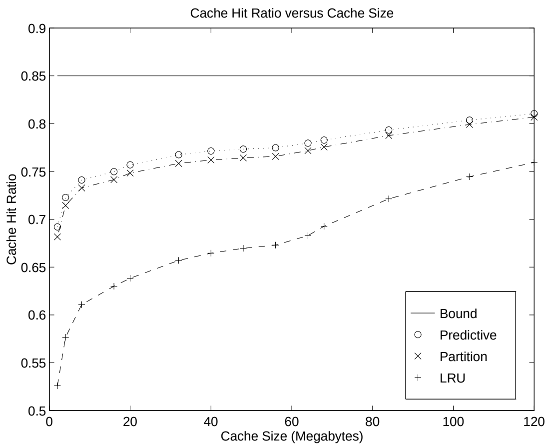

Figure 4.2: Average cache hit ratio versus cache size (2–120 megabytes, second order predictive models, prefetch threshold 0.1).

Table 4.1: Hit ratios for LRU and both predictive caches (cache size 4 megabytes, second order model, threshold 0.1).

Trace	A	B	C	D	E	F	G	H	Ave.
Predictive	56.1%	74.6%	75.0%	73.2%	72.1%	70.0%	77.7%	79.8%	72.3%
Pred. - Part.	*0.6%*	*0.5%*	*1.0%*	*0.5%*	*1.2%*	*1.0%*	*1.3%*	*0.8%*	*0.8%*
Partitioned	55.5%	74.1%	74.0%	72.7%	70.9%	69.0%	76.4%	79.0%	71.5%
Part. - LRU	*7.0%*	*14.4%*	*14.2%*	*15.5%*	*16.0%*	*15.0%*	*18.0%*	*10.2%*	*13.8%*
LRU	48.5%	59.7%	59.8%	57.2%	54.9%	54.0%	58.4%	68.8%	57.7%

To select this partition size of 16, we examined the relationship between partition size and cache hit ratio and found that a partition size of 16 or greater will capture the large majority of the predictive nature of a file within the Sprite traces. Figure 4.2 shows the cache hit ratio as compared to the partition size limit, (the partition size of 99 represents the full context model presented in Chapter 3 and thus an unlimited partition size). This static limit on the number of nodes in a partition means that we can treat each partition as a complete self-contained structure of a fixed size, significantly simplifying implementation. Restricting the partition size to 16 nodes or less will also enable us to use four bit pointers within a partition, so each node in a partition would require 11 bytes (file identifier, 8 bytes; counter, 2 bytes; pointer to first child and next sibling, 4 bits each), making the entire structure 176 bytes in size. Using this structure to model a four kilobyte object would impose an overhead of approximately 4.3%. More specifically, a four megabyte cache of four kilobyte blocks would require an active trie that was 176 kilobytes in size.

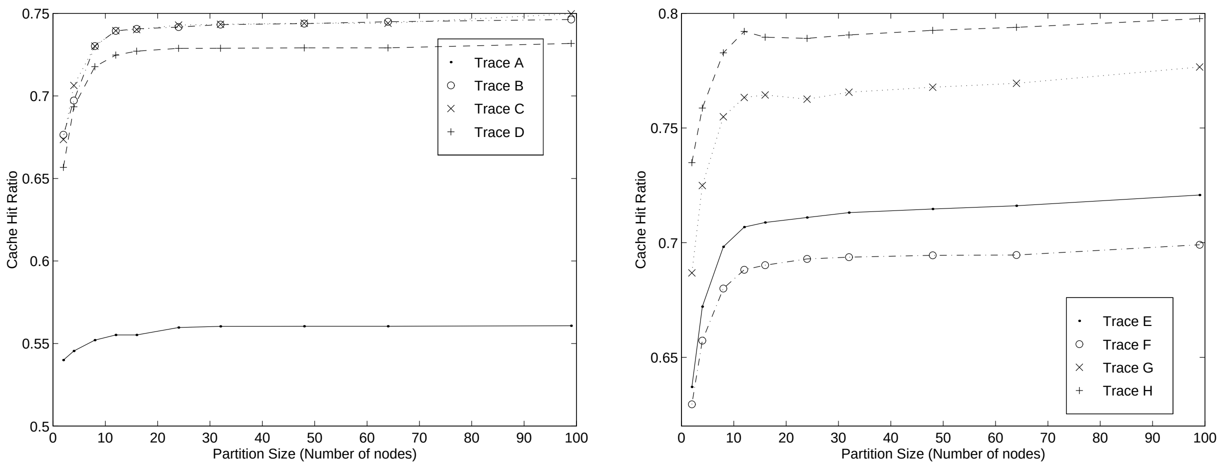

Figure 4.3: Partition size versus cache hit ratio (second order model, prefetch threshold 0.1, cache size four megabytes).

With this partitioned model we have traded a slight reduction in prediction accuracy for significant reductions in model space. In fact the maximum memory a partitioned model requires is less than one twentieth of the non-partitioned context model's buildup over a 24 hour trace. Additionally, over longer trace periods it is likely that the partitioned model will increase its prediction accuracy by adapting to areas of local activity.

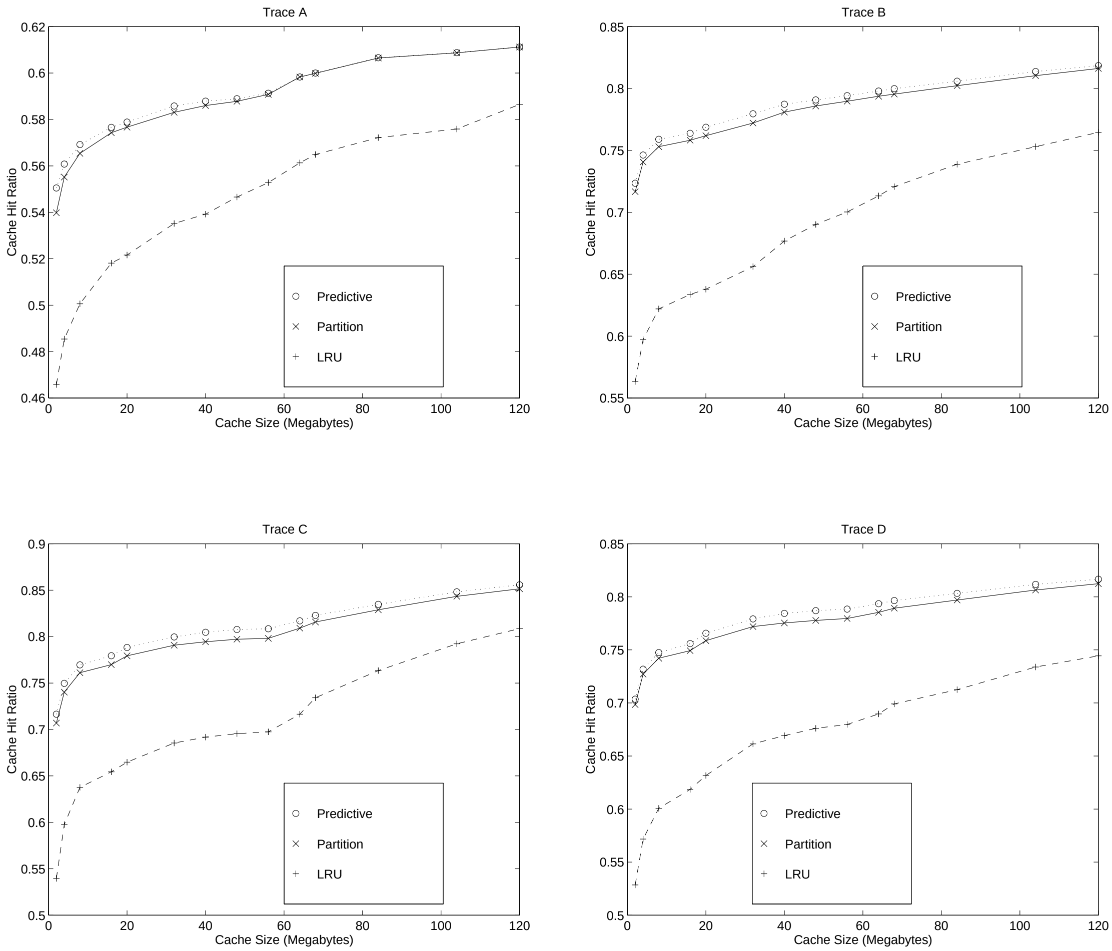

Figure 4.4: Cache hit ratio versus cache size for traces A through D (2–120 megabytes, second order predictive models, prefetch threshold 0.1, 16 node partitions).

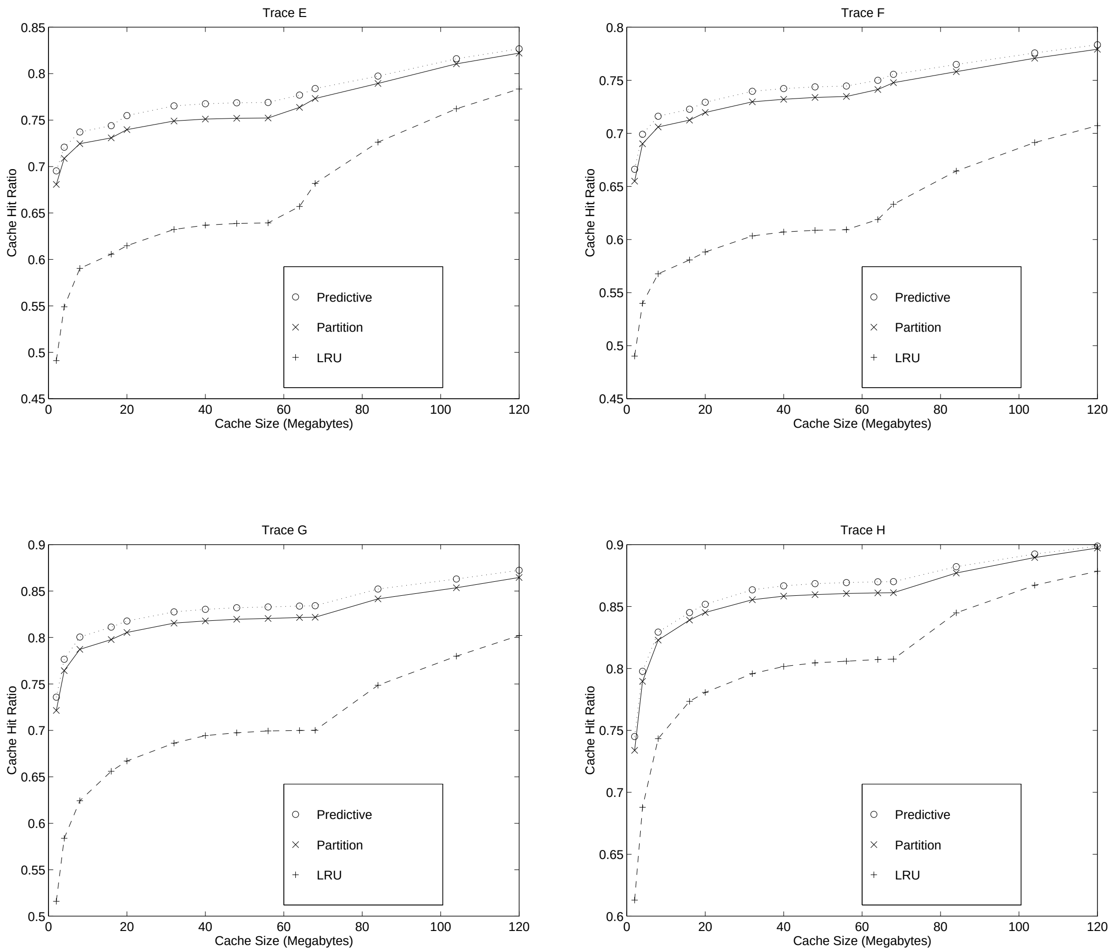

Figure 4.5: Cache hit ratio versus cache size for traces E through H (2–120 megabytes, second order predictive models, prefetch threshold 0.1, 16 node partitions).

Chapter 5

Related Work

While focusing on predictive caching, this work draws from several areas, ranging from discrete event prediction in machine learning and data compression to I/O system cache management and wide area network performance. In this section, we will attempt to highlight the research that has influenced our efforts.

5.1 Predictive Caching

Caching and prefetching have long been important topics in operating systems. However, using reference pattern predictions to combine these two techniques and improve cache performance is a relatively new and promising area of research.

Vitter, Krishnan and Curewitz [27, 28] were the first to examine the use of compression modeling techniques to track reference patterns and prefetch data. They prove that for a Markov source such techniques converge to an optimal on-line algorithm, and go on to test this work for memory access patterns [8] in an object-oriented database and a CAD System. Model size is dealt with by paging portions of the model to secondary memory. They also suggest that such methods could have great success within a variety of other applications such as hyper-text. Our work differs in the following ways. The reference patterns present in I/O systems represent an entirely different workload than those of memory accesses. We have addressed the issues of restricting model space and adapting to changing patterns. Our selection method differs in that we select based on a probability threshold, while they select the n most likely pages, where n is a parameter of their algorithm. Finally, we have adapted PPM in a different manner, without the use of vine pointers.

Within the domain of file systems, Griffioen and Appleton [9] have developed a predictive model that for each file accumulates frequency counts of the next n files (where n is a parameter of the algorithm). These frequency counts are then used to drive a prefetching cache. Their prediction model differs from ours in that they only consider a first order model and look at the next n events, while we consider multiple model orders and examine only the event immediately following. Nevertheless, they first presented the method of prefetch selection based on a probability threshold.

Lei and Duchamp [20] have pursued modifying a UNIX file system to monitor a process's use of the system calls **fork**, **execve**, **open**, **chdir** and **exit**. Using this information they build a tree that represents the access patterns of the process. Then as a process executes, its current access tree is compared with previously observed trees. If a match is found then the previous access tree is used to prefetch files. Since a current access tree must be compared with all those previously seen the computational complexity of this model limits its ability to scale to a large file system.

Laird and Saul [19] propose the use of *Transition Directed Acyclic Graphs* (TDAG) to learn the patterns in which items are requested from mass-storage devices and enable cache memories to minimize latency. They develop simulations based on a mass-storage system modified to use TDAG. Using a synthetic user workload they show that such techniques show great promise.

Kuenning, Popek, and Reiher [18] 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 *et al.* [17] have extended this work, 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. For mobile computing predictive caching has the potential not only to reduce latency but also to enhance disconnected operation by learning about relationships between files and prefetching related groups.

5.2 World Wide Web Caching

With the exponential growth of traffic and the large latencies that result from wide area distribution, cache performance has become a critical issue in the World Wide Web. With this understanding many researchers have looked for ways to improve current caching techniques. Padmanabhan and Mogul [23] have pursued using the model of Griffioen and Appleton to track requests at the server and then provide prefetching hints for the client. They show that reference patterns from a web server also offer useful information to support prefetching.

Bestavros *et al.* [5] have presented a model for the speculative dissemination of World Wide Web data. This work again shows that reference patterns from a web server can be used as an effective source of information to drive prefetching. These efforts have shown the value of reference pattern modeling within the World Wide Web, and emphasize the challenges that caching in the Web present. Current Web growth rates dictate that any predictive model must scale well or it will quickly become obsolete. Nevertheless the issues of model size and adapting to changing reference patterns are barely, if at all, addressed by previous efforts. Extending this work from the file system domain to the World Wide Web will provide an environment that will only increase the challenges posed by long term model buildup and changing reference patterns.

5.3 I/O Systems with Application Hints

Several researchers are exploring methods for cache resource management given application-provided hints. Patterson *et al.* [24] present an *informed prefetching* model that applies cost-benefit analysis to allocate resources. Cao *et al.* [6] examine caching and prefetching in combination and present four rules for successfully combining the two techniques and evaluates several prefetching algorithms to including an *aggressive prefetch* algorithm. Kimbrel *et al.* [13] present an algorithm that has the advantages of both *informed prefetching* and *aggressive prefetch* while avoiding their limitations.

While these methods offer significant improvements in latency and are of great value to scientific applications that process large data files in consistent patterns, they are 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 program.h**, while our predictive model could notice that every access to **program.c** caused an access to **program.h**. Further, an application-informed method would not be able to make use of relationships that exist across applications (*e.g.* **make** $\Rightarrow$ **cc**, **ld**).

5.4 Data Compression

Efforts in data compression strive to improve the computational complexity, adaptability and accuracy of content modeling techniques. Moffat *et al.* [21] addressed the model size by periodically constricting a new model with the last 2048 events and then using it to replace the current trie. More recently, Cleary *et al.* [7] have presented methods for extending PPM to support unbounded context lengths. In test cases these contexts rarely extended beyond a length of 10 characters. While this model improved accuracy and saw a 6% increase in compression, it also significantly increased computational complexity and model size. Improvements detailed in Chapter 6– specifically removing model order constraints and using the partition size limit to restrict model size and consequently model order– should enable this new model to exploit these higher order contexts, while reducing the computational complexity and model size. Investigating the effectiveness of a partitioned multi-order context model for text compression is one key area we have noted for future work.

5.5 Summary

Our work offers several contributions that have not previously been explored. To our knowledge this work is the first use of multi-order context models for file systems, and this is the first effort to address the issues of efficiently restricting model size and adapting to changing reference patterns. Nevertheless, the extensive amount of related work helps to make the point that there are many issues that still require further research.

Chapter 6

Future Work

While this work has shown that file system reference patterns provide valuable information for caching, and that such information can be tracked within reasonable resource constraints, we have also found many areas that offer potential for further gain. This chapter briefly surveys some of the areas intended for future exploration.

While cache hit ratios offer significant insight into cache performance, a more detailed simulation of I/O systems will offer us the ability to better understand the effects that predictive caching has on cache performance. The distributions for metrics such as read-wait, cache hits, overwrite rate, computational overhead, throughput and increased disk activity all require further investigation.

The partitioned model presented here is one successful method for efficiently restricting finite multi-order context models. Other variations of this model still require further exploration. Options such as using other metrics like recency to maintain partition size and prefetch selection present one option for improvement. Another possible improvement is partitioning the trie based on some set of nodes (representing a set of reference patterns) other than simply all the first order patterns. For example, one approach could be to select some set of reference patterns and statically limit the number of nodes descended from the nodes representing those patterns. A frequently accessed files such as **make** could have separate partition for its four most frequent children and then one partition for the rest.

Since this new model limits the number of descendants of any node, excluding the root node, can we remove the limit on model order? Since the partition size will constrain our model it can also serve as a limit on the maximum order modeled. This change could significantly increase the flexibility of this model. It would allow partitions to extend linearly (increasing in depth) for reference patterns that consistently repeated in the same sequence, and laterally (increasing in breadth) for those reference patterns that contained greater variation.

While the system presented here uses a model to prefetch files, it still uses LRU to determine which files to expel from the cache when space is needed. It is quite possible that these or similar models could be used to affect replacement policies as well.

Finally, the success of our partitioned context model for predicting file system events leads us to question how well such a predictive model would do for text compression.

Chapter 7

Conclusions

This work offers the following new contributions. We have shown that well-known context modeling techniques are effective in modeling the reference patterns of a file system to drive prefetching. We have also shown how such techniques can be improved to work in reduced model space and adapt to changing reference patterns.

Our ability to effectively predict future events based on previous file reference patterns provides strong empirical evidence to support our original hypothesis: that there exist consistent and exploitable relationships between file references (*e.g.* **make** $\Rightarrow$ **cc**, **ld** ...). Therefore, conventional caches, which ignore reference patterns, are failing to make full use of the information available. Our simulations have shown that prefetching based on previous file reference patterns offers a performance increase that cannot be easily matched by increasing the cache size.

We have also demonstrated that for any model to be realistic it must address the issues of model space constraints and of adapting to changing reference patterns. By partitioning the context model and limiting partition size, we have presented a method that addresses these issues. This partitioned model closely approximates the accuracy of an unpartitioned context model, while working in linear space with respect to the alphabet size, and constant run-time.

With the continued growth of processor speeds and new performance challenges from the World Wide Web the performance bottle-neck presented by I/O will only increase. By exploiting the highly related nature intrinsic to computer systems, methods such as predictive caching enable a computer system to manage resources with more complete information, dramatically improving cache performance and reducing the I/O bottleneck.

Bibliography

[1] Abdulla, G., Abrams, M., and Fox, E. A. Scaling the WWW. Tech. Rep. TR-96-06, Virginia Polytechnic Institute and State University, Department of Computer Science, March 1995.

[2] Almeida, V., Bestavros, A., Crovella, M., and de Oliveira, A. Characterizing Reference Locality in the WWW. In *IEEE PDIS'96: The International Conference in Parallel and Distributed Information Systems* (Miami Beach, Florida, December 1996), IEEE Computer Society.

[3] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed file system. In *Proceedings of Thirteenth Symposium on Operating Systems Principles* (August 1991), Association for Computing Machinery, pp. 198–212.

[4] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[5] Bestavros, A., and Cunha, C. A prefetching protocol using client speculation for the WWW. Tech. Rep. TR-95-011, Boston University, CS Dept, Boston, MA 02215, April 1995.

[6] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 SIGMETRICS* (May 1995), Association for Computing Machinery.

[7] Cleary, J. G., Teahan, W. J., and Witten, I. H. Unbounded length contexts for PPM. In *Data Compression Conference* (March 1995), IEEE Computer Society, IEEE Computer Society Press.

[8] Curewitz, K. M., Krishnan, P., and Vitter., J. S. Practical prefetching via data compression. *SIGMOD Record 22*, 2 (June 1993), 257–266.

[9] Griffioen, J., and Appleton, R. Performance measurements of automatic prefetching. In *Parallel and Distributed Computing Systems* (September 1995), IEEE, pp. 165–170.

[10] Gwertzman, J., and Seltzer, M. The Case for Geographical Push Caching. In *Fifth Annual Workshop on Hot Operating Systems* (Orcas Island, WA, May 1995), IEEE Computer Society, IEEE, pp. 51–55.

[11] Gwertzman, J., and Seltzer, M. World wide web cache consistency. In *Proceedings of the USENIX 1996 Annual Technical Conference* (San Diego, CA, Jan 1996), USENIX, USENIX, pp. 141–152.

[12] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *Transactions on Computer Systems 6*, 1 (February 1988), 51–81.

[13] Kimbrel, T., Tomkins, A., Patterson, R. H., Bershad, B., Cao, P., Felton, E. W., Gibson, G. A., Karlin, A., and Li, K. A trace-driven comparison of algorithm for parallel prefetching and caching. In *Proceedings of Second USENIX Symposium on Operating Systems Design and Implementation* (October 1996), USENIX.

[14] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the coda file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles* (October 1991), Association for Computing Machinery, pp. 213–25.

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

[16] Kroeger, T. M., and Long, D. D. E. Predicting file-system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference* (January 1996), USENIX.

[17] Kuenning, G. The design of the seer predictive caching system. In *Workshop on Mobile Computing Systems and Applications* (December 1994), IEEE Computer Society, pp. 37–43.

[18] Kuenning, G., Popek, G. J., and Reiher, P. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of USENIX Summer Technical Conference* (1994), USENIX, pp. 291–303.

[19] Laird, P., and Saul, R. Discrete sequence prediction and its applications. *Machine Learning* (1994).

[20] Lei, H., and Duchamp, D. An analytical approach to file prefetching. In *Proceedings of USENIX 1997 annual Technical Conference* (January 1997), USENIX.

[21] Moffat, A. Implementing the PPM data compression scheme. *IEEE Transactions on Communications 38*, 11 (November 1990), 1917–1921.

[22] Ousterhout, J. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer Technical Conference* (June 1990), USENIX, pp. 247–56.

[23] Padmanabhan, V. N., and Mogul, J. C. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the 1996 SIGCOMM* (July 1996), Association for Computing Machinery, Association for Computing Machinery.

[24] Patterson, H., Gibson, G., Ginting, E., Stodolsky, D., and Zelenka, J. Transparent informed prefetching. In *Proceedings of thirteenth Symposium on Operating Systems Principles* (December 1995), Association for Computing Machinery.

[25] Trivedi, K. *Probability & Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[26] van Renesse, R., Tanenbaum, A. S., and Wilschuts, A. The design of a high-performance file server. In *Proceedings of the Ninth International Conference on Distributed Computing System* (1989), IEEE Computer Society Press., pp. 22–27.

[27] Vitter, J. S., and Krishnan, P. Optimal prefetching via data compression. In *Proceedings 32nd Annual Symposium on Foundations of Computer Science* (October 1991), IEEE Computer Society Press, pp. 121–130.

[28] Vitter, J. S., and Krishnan, P. Optimal prefetching via data compression. *Journal of the ACM 43*, 5 (September 1996), 771–793.

[29] Williams, S., Abrams, M., Standridge, C. R., Abdulla, C., and Fox, E. A. Removal policies in network caches for world-wide web documents. In *Proceedings of the 1996 SIGCOMM* (1996), SIGCOMM, Association for Computing Machinery.