

File Prefetching for Mobile Devices Using On-line Learning

Zaher Andraus, Anthony Nicholson, Yevgeniy Vorobeychik

Electrical Engineering and Computer Science

University of Michigan

{zandrawi, tonymich, yvorobey}@eecs.umich.edu

Abstract—As mobile, handheld devices have increased in power and function, users increasingly treat them not as mere organizers, but as an extension of their personal computing environment. But accessing files remotely can lead to poor performance, due to network latency. While caching can alleviate this problem, existing caching schemes were designed for desktop clients, and are not mindful of the network and storage limitations endemic to mobile devices.

Much work has been done on *prefetching* files based on various predictive methods. While each method may do well for its target usage pattern, performance can quickly degrade as usage patterns change. Our insight is that one can get the “best of all worlds” by distilling suggestions from varied predictors into one final prediction. Our system learns how each property performs over time and dynamically adjusts the weight of each in the final decision. The resulting system is simple and robust, and supports an arbitrary number of properties, which can be plugged in or out at the administrator’s discretion.

Our implementation within the Coda file system shows a 10-15% reduction in average file access latency for low-bandwidth, high-latency connections to the file server, a common usage scenario for many mobile users.

I. INTRODUCTION

Disk and network latency bottlenecks have been on the minds of systems researchers for many years, and caching has long been used to alleviate this problem. Caching is clearly most effective when the client-side cache can be large enough to hold the common working set of user files, necessitating fewer fetches on demand over the network. As file sizes increase over time, desktop and laptop users can maintain distributed file system performance simply by increasing the size of their local caches. Storage is plentiful, as multi-gigabyte hard drives are now the norm.

Mobile, handheld devices, on the other hand, do not always have this luxury. PDAs commonly have a storage capacity on the order of megabytes, not gigabytes. While this can be augmented by additional storage (such as an IBM MicroDrive), the added bulk and expense make that an unattractive option to many users. Also, keeping the

cache in RAM instead of on an external disk can greatly extend precious battery life by minimizing disk spin up and down operations.

The performance penalty for cache misses is also higher in a mobile environment. By definition, mobile devices utilize wireless links, which may range from fairly high-speed (e.g. 802.11) to low bandwidth, high latency links (e.g. cellular data access). While the penalty for a cache miss is not severe for a desktop client, which is usually connected to the file server via a 10 or 100 Mbps connection, it can be a major consideration for mobile clients.

Cache performance improvement can be approached in two ways: one may either try to improve the cache replacement strategy, or to make better decisions regarding which files should be cached. Both have been explored, albeit independently, by a number of researchers. We concentrate on the latter. More specifically, we have developed an algorithm that predicts future file accesses from different parameters, or *properties*, using on-line learning. Our algorithm dynamically learns the effect that various properties have on successful file access prediction and adjust their relative importance accordingly. If the environment (a user’s working set) does not change very frequently, allowing enough time for the algorithm to learn the access patterns, it should perform quite well. This assumption is supported by Kuenning’s findings [9], [10], [11].

Most previous attempts at intelligent file prefetching (see Section II) focused on one main criterion, such as sequence of previous accesses, semantic distance between files, or directory membership. While each of these may work well for certain usage patterns, they cannot adapt to behavior which doesn’t fit their *a priori* worldview. Our insight is that it should be possible to run many varied predictors simultaneously, and let the system decide, based on past performance, which predictor properties are returning the best results. This flexibility is one of our system’s most novel aspects. Different properties may be designed to work well in very specific settings, but collectively the system should yield good results for a wide range of usage situations.

Another major strength of our system is its modular design and the simple interface between the algorithm module, which performs all the organizational operations and the actual learning, and the properties, which collectively predict a set of files to be prefetched. The properties are completely isolated from each other, and are not aware of each other's existence. The algorithm itself accesses all the properties through a uniform interface and does not care about their individual identities. This level of isolation is precisely what allows us to add and remove the properties at will.

We have implemented our system within the Coda distributed file system, through only a small stub to the Coda source code. The remainder of our system is platform-independent, allowing easy portability to various file and operating systems¹.

Section II discusses the various existing approaches to file prefetching in the literature. Section III delves deeply into the design and implementation aspects of our system, including a description of the properties we have implemented. Performance evaluation and results are detailed in Section IV, and Section V concludes.

II. RELATED WORK

One of the earliest ideas for file prefetching was to utilize application hints that specify future file accesses. This deterministic prefetching was explored by Patterson et al. in several articles on *Informed Prefetching and Caching* [17], [20]. While this technique provided some important insights into the tradeoffs of prefetching, it is not very generalizable, as few applications produce the required hints.

At around the same time, the SEER project was born at UCLA [9], [10], [11]. The goal of SEER was to allow disconnected operation on mobile computers using automated hoarding. A rather successful attempt was made to group related files into clusters by keeping track of *semantic distances* between the files and downloading as many complete clusters as possible onto the mobile station as can fit into the cache prior to disconnection. They defined semantic distance between some two files, A and B, as the number of references to other files between adjacent references to A and B. Subsequent versions also incorporated directory membership, "hot links", and file naming conventions into the hoarding decision process.

Appleton and Griffioen [4], [5] used a directed graph, the nodes of which represented previously accessed files, with arcs emanating from each node to the node (file)

that was accessed within some lookahead period afterward. The weight of each arc was the number of times it had been visited (i.e., the number of times the second file was accessed within the lookahead period of the first). Thus, if some file was accessed, the probability of some other file being accessed "soon" can be estimated from the ratio of the weight of the arc to that file to the cumulative weights of all arcs leaving the current file.

Kroeger [7], [8] used a multi-order context model implemented using a trie, each node of which represented the sequence of consecutive file accesses from the root to that node. Each node kept track of the number of times it had been visited. Slightly reminiscent of Appleton and Griffioen's model, the children of each node represented the files that have in the past followed the access to that file. The probability of each child node being the next victim can be estimated from the ratio of its visit count to the visit count of its parent less one (since the parent's visit count has just been incremented). In a later work [8], Kroeger enhanced his model by partitioning the trie at its first level and maintaining a limit on the size of each partition.

Several other projects have subsequently tried to improve on these efforts. The CLUMP project [2] attempted to leverage the concept of semantic distance developed as a part of the SEER project to prefetch file clusters. Lei and Duchamp built a unique probability tree similar to Kroeger's for each process [12]. Vellanki and Chervanak revisited the Patterson's Cost-Benefit analysis, but adapted it to a probabilistic prefetching environment [21]. Geels unsuccessfully attempted to use Markov Chains for file prefetching [3]. Finally, an adaptive cache replacement algorithm that uses learning techniques was presented by Ari et al. [1]. This adaptive algorithm is the closest model to ours that we have found in literature. The main difference between their work and ours is that we concentrate on prefetching, whereas their algorithm deals with replacement strategies.

All the prefetching models that we have seen only concentrate on one prediction method, and, thus, our project can be seen as an extension to the efforts mentioned above. We combine the probability trie proposed by Kroeger and probability graph per Appleton's work with several other properties which one would intuitively expect to be good predictors of future accesses, such as file extension and directory membership. Given all of these potential predictors, we developed an algorithm that learns their relative importance in a given environment. Indeed, we feel that one of the main shortcomings common to all the approaches to date is their inability to perform well in changing environments. Our system allows a set of predictors to dominate the prefetching

¹In fact, the algorithm module and all properties have been compiled and tested both under Linux and Microsoft Windows 2000.

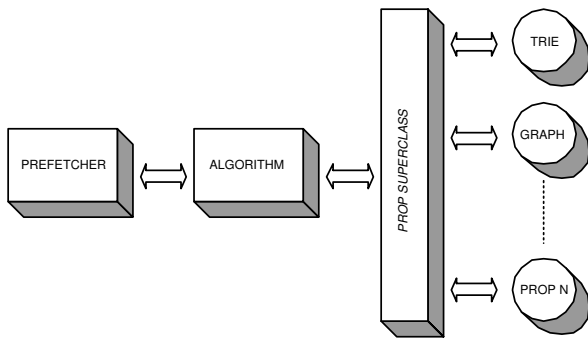


Fig. 1. System Design

decisions in the environments to which they are best suited, and to give way to others when those become more successful.

The topic of combining multiple predictors is already well established in the learning community. The Rosenblatt algorithm [18] as well as neural network algorithms are robust building blocks for classification, learning and prediction. Most of these algorithms are linear-threshold algorithms, making them fast and easy to implement. These methods make predictions based on the weights of predictors, and these weights are refined every time a prediction errs. The Weighted Majority Algorithm [14] is yet another way to combine predictors. It works well when the predictors are experts, i.e. output a prediction every time it is requested. The main difference between the Weighted Majority Algorithm (WMA) and Rosenblatt based schemes is that in the WMA the weights are updated in every prediction. The Winnow Algorithm [13] is similar, but combines specialist properties (not necessarily experts), which have the right to abstain and return no prediction. For our system, “abstaining” will mean returning an empty prediction list.

All of the above methods previously have been theoretically and empirically evaluated. However, they all return a single prediction, whereas we are trying to predict a number of files to prefetch each time. This difference led us to define the problem in terms of allocating the cache between predictors. Thus, our approach can be considered as combining *multi-file* prefetching based on a *single-predictor* [2], [8], [11] with *single-file* prefetching based on *multiple-predictors* [13], [14], [18]. We implemented both Rosenblatt-based and specialists-based algorithms. Since it was unclear which would perform best, we evaluated our system’s performance while driven by variations of both.

III. DESIGN AND IMPLEMENTATION

A. Design Overview

As our system aims to perform network file prefetching within the context of a Distributed File System, the choice of one was a critical design decision. We ultimately chose Coda [19] as our framework. Coda is an attractive choice for several reasons. It was developed primarily as an academic research tool, and is often used/cited in research, providing us with a source of file traces and examples of valid measurement techniques [16]. It has also been ported to many combinations of hardware and operating systems. This dovetails nicely with our goal of keeping OS-specific code to a minimum. If we are designing our system to be portable, it follows we should choose a base DFS that can follow us where we want to go.

As shown in Figure 1, we have broken our logic into several parts. The primary motivation for this was portability. Only the Prefetcher module interfaces with the native OS or DFS, and would therefore be the only code to change in order to port our system to new systems. The entire system is compiled into the Coda client-side cache manager, Venus. By linking our source into Venus we reduced communication overheads to that of a function call.

The Prefetcher module is invoked at critical points in Venus execution, such as when servicing a file open request, or when about to fetch a file from network. It keeps the higher-level modules informed of file activity, and acts on the prefetching suggestions provided by the Algorithm Module.

The Algorithm Module consists of the Algorithm logic (the block labeled “Algorithm” in Figure 1) and various property data structures.

B. Prefetcher Module

In order to successfully implement prefetching within the context of the Coda distributed file system, we need to be able to:

- 1) Monitor all open calls to files which are part of the mounted network volume(s), and
- 2) Monitor all fetch-from-server operations, so that our logic may suggest appropriate files to prefetch at that time.

We can accomplish both these goals through modification of the Coda client software alone, which simplifies our development effort. Even better, the code which we are concerned with on the client all resides in the cache manager Venus, which is a user-level process. The fact that we do not have to modify the kernel greatly

simplifies the porting of our system to other operating systems.

We have defined a C++ class, `prefetcher`, which encapsulates the functionality required to perform intelligent, adaptive prefetching. An instance of this class is a member of Venus' `class fsdb` (which defines the file system thread).

At the point where Venus receives an upcall from the kernel requesting a file open, `prefetcher` observes this and notifies the Algorithm Module of the file which was accessed, so that it may in turn inform the properties. Once the FS thread receives an open request, it first checks if the file is already in the Coda cache. If not, it issues a `Fetch()` request to retrieve the file from the server. At this point, the Prefetcher calls the Algorithm Module, with the file name currently being accessed. It expects in return a list of files which are suggested for prefetching at this time. The Prefetcher parses the list and removes those files from the list that are already resident in the cache (we get those for free!). The remaining files the Prefetcher "prefetches" from the server, using the standard `Fetch()` call. Therefore, the standard Venus code is not aware of which files are being fetched due to a legitimate cache miss, and which fetch requests are a result of actions of the Prefetcher. Clearly, the Prefetcher must keep track of which requests it generated and not invoke the prefetch logic on those to avoid an endless loop.

C. Algorithm Module

1) *Decision-Making in the Algorithm Module:* The algorithm module decides which files are likely to be accessed in the near future. As it is expected to manage all data necessary for the prediction, it is notified of all file accesses synchronously² and passes on this information to each property. The prefetcher informs the algorithm module whenever Coda is about to perform a remote file fetch, so that it may recommend other files appropriate for prefetch at this time (and thus amortize the cost of a remote fetch over several files rather than just one).

To make this decision, the Algorithm Module itself relies on a set of *properties* to make their individual predictions. One can think of the set of properties as predictors running in parallel, unaware of each others' existence.

But how does the algorithm module decide who to believe? We decided to have the algorithm module track the past performance of each property, and adjust the

relative weights of each accordingly. As the prefetcher monitors cache hits and misses, our system can determine if a prediction was too aggressive (file was prefetched, but never referenced) or too conservative (cache miss on a file which should have been prefetched). While all properties are created equal, they soon may diverge in importance, as some are observed by the global decision-making unit to be "weak" predictors. Thus, the algorithm module is analogous to a president, who delegates decisions down to advisers, with some advisers given more trust than others as a consequence of their past service. The final decision, of course, is left up to the global decision-maker, which evaluates each *potential* file from the pool of all files and selects the subset that should be prefetched.

We attempted to answer the following questions during the algorithm design process:

- Which properties should be used to rank files?
- How do the properties rank files?
- How do we determine the final list of files to prefetch?

The answer to the first question is still open, as there is a very large number of file properties which could be potential predictors of access patterns. Instead of setting our choices in stone, we created a modular architecture that would allow additional predictors to be simply "plugged-in" into the algorithm.

Each property exports the following simple interface to the Algorithm Module:

- *file_accessed(file)*: notify the property of a file access event
- *get_prefetched_list(size)*: ask for a list of predicted files, which would fill up to *size* bytes of the cache

The list of prefetched files that is returned must be sorted by priority that the respective property assigns to it. We refer to the relative position of a file within this list as the *ranking* of this file with respect to the property.

So, how do the properties rank files? We tend to leave this decision up to the properties, with one constraint: the ranking should be an indicator of importance that the property attributes to the file, with the importance decreasing as one moves from the head to the tail of the predicted file list. The rankings of the files in the returned list are normalized to $0 \leq r \leq 1$ by substituting the index of the file within the list into the function $r = f(p) = \frac{1}{p+1}$ when $p = 0, 1, \dots$ is the index.

2) *On-Line Learning and Combining Predictions in the Algorithm Module:*

a) *On-Line Learning:* So, how does the algorithm module reconcile all the information contained in the properties? Our answer is on-line learning. More pre-

²This is important because some properties (for example, the trie) need to know the exact sequence of file accesses.

```

for (each_property) {
  if (cache_miss) {
    property → get_prefetch_list(entire_cache)
  }
  rank_of_file = property → notify_file_accessed(file)
  if (cache_miss) {
    weight[property] = UpdateWeight(property, rank_of_file)
  }
}

```

Fig. 2. Computing weights. UpdateWeight function is implemented as a part of the specific learning algorithm.

cisely, we learn the weights that will be used in combining the predictions from different properties. This is just like a reputation system: if we trust a property, we will be more likely to listen to its predictions.

We decided to explore two learning approaches: the linear-threshold method (Rosenblatt) and the Winnowing algorithm. The linear-threshold algorithm [18] reacts to counter-examples (misses) by adding the current input to the weights; otherwise, the weights remain unchanged. We used this algorithm when the input is the ranking of the accessed file by each property. This policy leads to boosting the weights of properties that considered the missed file more important.

The Winnow algorithm also reacts to misses. After each miss, the weight of the property is adjusted in the following way:

- if it abstained the last time it was called, do not change its weight
- if it predicted a list of files that contained the current file, promote its weight by multiplying it by a constant $\alpha > 1$. If the list doesn't contain the correct file, punish the property by multiplying its weight by $0 < \beta < 1$.

As you can see, this original version of Winnowing ignores the ranking of the file in the predicted lists of the properties that had a hit. In a hybrid version of the Winnow algorithm, the promotion factor depends on the place of the file in the successful lists. As part of our evaluation, we sought to determine which method provides the best performance.

In general, the weights of all the properties are updated as follows (see pseudocode in Figure 2):

- 1) weights are only updated on cache misses
- 2) the ranking of the file is calculated based on the emulation call to get all prefetching suggestions that would fit into the entire cache, since this is the maximum amount of space any property can ever service
- 3) the ranking is calculated based on position, p , in the list returned by the

get_prefetch_list(entire_cache) call from the formula $\text{ranking} = \frac{1}{1+p}$, which is just an arbitrary decreasing function in p

Irrespective of the method, we have to determine how to combine the predictions from different properties based on their weights. We discuss our approach to this in the next subsection.

b) Combining Predictions: At this point, we had several options for using the learning scheme just described to combine predictions made by the properties into one list. A traditional approach would have been to calculate the overall ranking as $\sum_{p \in \text{properties}} w_p \cdot r_p$ and to use it, or a revised monotonic transformation of it, to check whether it is sufficient to fill the target size the prefetcher specified. Another approach is to divide the available prefetching storage space according to the weights, and allow each property to use its proportion of the total space as its own cache to fill with predicted files. We chose the latter approach, referred to as Size Division, for several reasons. First of all, it is fairer toward the properties with small but significant weights. We are concerned about such properties, since they may become significant in unstable situations, such as a change of the working set. Additionally, Size Division allowed us to leave the ranking decisions completely encapsulated within the properties, making the primary decision-making of the algorithm simpler and more general. The downside of the Size Division method is that it requires more computation from the module, which will have to deal with the overlap between file sets returned by the properties. It also punts much of the complexity onto the properties. We think this is acceptable because properties are inherently more volatile units which may be changed a number of times, while the body of the Algorithm Module is stable. Furthermore, the property/algorithm interface is greatly simplified, facilitating dynamic adjustments that may often be made to the properties, as well as the process of adding and removing properties.

Thus, our algorithm uses weights to decide on the proportion of the cache it allocates to each property. This calculation is straight forward: a property gets to prefetch a portion of the cache proportional to its weight divided by the sum of all weights.

3) Properties: In the previous sections we have alluded to the properties that perform most of the thinking for the algorithm module, but have thus far been very vague in describing what they are. The following subsections describe the seven properties that we have implemented.

a) Trie: The trie property was created using a multi-order context model described by Kroeger [7], [8]. We chose to use second order context, since the size of

the trie is exponential in the context order, and Kroeger showed no improvement in predicting ability beyond second order.

The insight of this property is that it is very likely that many applications or utilities access the same sequences of files at different times. A good example is a development environment, where a Makefile will tend to compile files in the same sequence.

An interesting problem we ran into during the implementation of the trie is that of determining the best files (according to cumulative probability) to place in a fixed sized space. This problem turns out to be NP-Complete³, and, therefore, we used a heuristic that placed as many files as possible into the fixed size, ordered by probability of future access, and then iteratively replaced the last file with a number of smaller files with higher cumulative probability.

After having implemented the trie, we found that the overhead imposed by storing the complete history of file accesses is unacceptable. As can be easily seen, the space requirements of a trie of context m to store a database of n accesses is $O(n^{m+1})$, so in our case it is $O(n^3)$. As n grows over time, adding files to the database, as well as retrieving predicted file lists becomes slow. To remedy this problem, we followed Kroeger’s example [8] and implemented constant partitions. Indeed, our results in Section IV-A.2 show that varying partition size had a significant affect on the trie overhead.

b) Probability Graph: This is exactly the probability graph proposed by Appleton and Griffioen [4], [5]. The graph stores each file access as a node and tracks subsequent file accesses, recording the number of times a given file was a successor. Successor relationships are represented by directed arcs in the graph, and access counts are recorded as the weights of these arcs. When the `get_prefetch_list` method is invoked, the property returns all files that succeeded the currently accessed file within a specified *lookahead window*. This lookahead window can be seen as the access distance, which is analogous to the semantic distance used in SEER.

c) Last Successor: This property relies on long-term temporal locality of file accesses. In other words, it “records” the immediate successor of each accessed file and, when asked, releases this object to the algorithm. While this is the simplest property that we deal with, intuitively it should be fairly effective, as we would expect people to often follow the same working patterns. This, naturally, is subsumed by the Trie and Probability Graph properties, which not only maintain the last successor, but a set of successors (thus, providing a probability

distribution of future successors, as opposed to a point estimate).

d) Directory Distance: Directory Membership property tries to relate file system locality to temporal locality of access, since files that reside in the same folder are likely to be a part of the same working set. This property keeps track of the directory in which the accessed file resides and its predictions are based on directory distance. Distance of 0 between two given files indicates that they are in the same directory; distance of 1 means that one file lives in the parent directory of the other, and so on. Predictions are made by following the directory hierarchy and ranking files according to directory distance.

e) Directory Probability Graph: This property maintains a successor graph of directories in the same way as the Probability Graph property described above maintains a successor graph of files.

f) Directory LRU: Directory LRU maintains a FIFO queue of directories. When a directory is accessed, it is added to a queue, possibly displacing the Least Recently Used directory. When returning the prefetching suggestions, it follows the queued directories from the top of the queue, ranking files accordingly.

g) File Extension: Intuitively, this again seems like a good indicator of access locality, as it is easy to envision applications such as a compiler or MP3 player accessing many files of same extension one after another.

IV. PERFORMANCE EVALUATION

Our test setup consisted of two Dell Latitude laptops, both with Pentium III 900 MHz processors, 512 MB RAM, and 10/100 Mbit/s Ethernet cards. While the laptop used as the client is certainly more powerful than most handheld devices, this should not affect our results, given that network latency and cache size are the limiting factors in these tests. We used a Linux kernel module (NistNet) to simulate various network bandwidths between client and server. Both test machines ran Linux kernel 2.4.18-3 and Coda software release 5.3.19. The Coda server software is unmodified; the client was running our modified Venus cache manager as described above.

We identified two scenarios we wanted to evaluate. First, we obtained traces of actual file accesses from a client at Carnegie Mellon University, and replayed these traces to compare the performance of our modified system to baseline Coda, and to a Trie approach (Kroeger’s solution). Second, we ran a more contemporary benchmark by placing the source tree of the Apache web server in Coda, and measuring the time to build (forcing fetch

³It can be reduced from the Knapsack problem.

of source files over the network) for a range of network latencies.

A. Evaluations using Coda Traces

The following tests were performed with file access calls simulated from Coda traces collected on the mozart computer at Carnegie Mellon University in 1993 for a period of about one month. This particular machine was chosen because it is described as being a “typical” workstation. Based on the information in the traces, we regenerated all the files with their original file sizes (but filled with random data). As these traces are 10 years old, the file sizes are much smaller than what would be common today. If we used a cache size, therefore, which would be reasonable for a mobile user today (such as 15 or 30 megabytes), the entire working set touched in the run of the traces might fit in the cache, negating most of our system’s effectiveness. Instead, we used a small cache size of 4 MB. As the ratio of cache size to working set size is a critical determining factor in cache performance, we believe this small cache with these small old traces will give us a similar performance evaluation of our system as a larger cache would in tandem with contemporary, larger-sized files.

One notable shortcoming in the Coda traces was the lack of file size information for files that were never opened during the tracing period. Since some of our properties may prefetch these files, we had to assign some realistic sizes to these files. We did this by following a depth first search through the file hierarchy and assigning, to any zero-byte files, the last size encountered. While we understand that this is not statistically the most appropriate solution, we felt that since we will still mostly prefetch files with known size, this workaround would not have much impact on the results.

1) *Network Latency Evaluation:* We hypothesized our system would be more beneficial as network latency to the file server increases, as whatever computational overhead our system has introduced should be overshadowed by the network delay to fetch a file. Any overhead would essentially be hidden by natural idle periods between file fetches in the traces, and these idle periods would grow longer as network latency to server increases.

We ran the Coda traces described above, for the following Prefetcher configurations:

- no properties active (baseline Coda behavior)
- Trie only (Kroeger’s method [7])
- all properties, linear threshold (Rosenblatt) algorithm
- all properties, Specialists Winnow algorithm
- all properties, Specialists Hybrid algorithm

Bandwidth	Trie	SpecialistsWinnow	SpecialistsHybrid
10 Mb/s	7.92%	-5.75%	-2.67%
1 Mb/s	-5.99%	-12.1%	-18.89%
500 Kb/s	9.15%	4.62%	10.59%
100 Kb/s	3.12%	23.65%	-9.23%
56 Kb/s	7.65%	10.16%	-9.34%
28.8 Kb/s	11.64%	14.46%	11.29%

TABLE I

PERCENTAGE IMPROVEMENT IN TRACE PLAYBACK TIME BY ALGORITHM METHOD. (a negative number indicates trace playback time increased)

For each test listed above, we used the network latency simulator to simulate a range of network conditions between the client and server, corresponding to typical mobile usage scenarios. For example, 10 Mb/s and 1 Mb/s would correspond to typical (low) latencies for on-campus wireless LAN access, 500 Kb/s and 100 Kb/s to access to the file server from off-campus via DSL or a cable modem, and 56 Kb/s and 28.8 Kb/s to wireless WAN access, through a slow cellular link. The test script referenced 45685 files. All tests were repeated three times, and the average used.

Figure 3 shows the average access latency vs. bandwidth, for each prefetcher configuration. Average access latency is calculated as the total time required to replay the entire Coda trace set, divided by the total number of file accesses. We have omitted the results for the Linear Threshold Algorithm, as they turned out to be very poor (twice as bad as the other three configurations). From post-examination of our traces and logs, it is clear that the Linear Threshold Algorithm did not converge fast enough to provide benefit. That is, it acted on far too many poor recommendations because it did not adjust the relative property weights quickly enough. As one can see, however, the more aggressive Winnowing scheme generated far better results.

The results show that our system does not provide much benefit for LAN speeds (1-10 Mb/s), but as bandwidth drops and latency to the file server increases, our system begins to out-perform the base Coda implementation. One can conclude that on the high bandwidth runs our poor performance is due to our inherent system overhead. As file access latencies increase, this overhead makes up less of the total run time, and effects of our prefetching causes enough cache hits to make the difference. Our SpecialistsWinnow implementation is consistently the best, beating unmodified Coda’s performance for all bandwidths 500 Kb/s and below. Its success is especially pronounced for the tests with the least amount of bandwidth and highest latency to server–

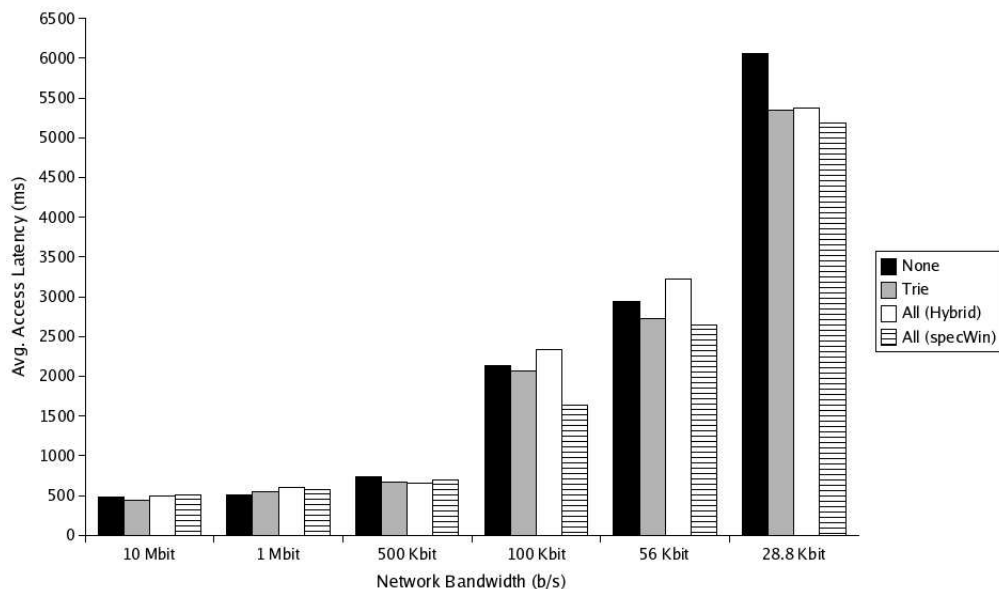


Fig. 3. Avg. Access Latency (ms) vs. Network Bandwidth to Server

our system beats the baseline by 10% for the 56 Kb/s tests, and almost 15% for the 28.8 Kb/s tests. These lower bandwidths correspond to our target application (weakly connected mobile devices).

We also compared our performance to an optimized version of Kroeger’s trie, to shed light on how our system stacks up to a proven method from the literature. The results show our system outperforms Kroeger’s trie for 100, 56, and 28.8 Kb/s connections, and for higher bandwidths the performance is comparable.

2) *Trie Overhead*: Our ultimate goal is to make this system adapt optimally to changes in network speed, and a necessary step is to learn how our parameters affect both latency and overhead. This information can be later used to create algorithms that maximize the end-user’s utility, which we assume to be an inverse function of average file access latency per byte.

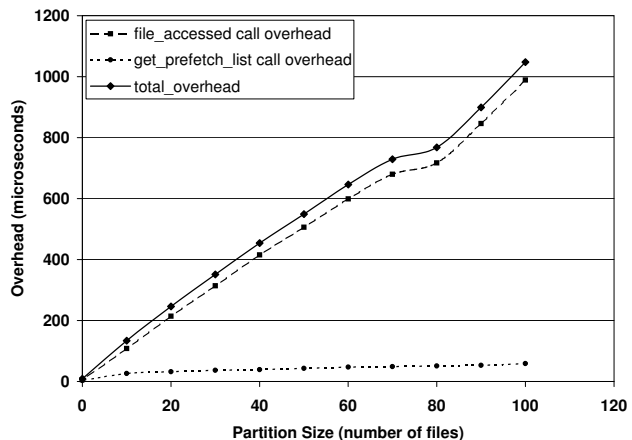


Fig. 4. Trie Overhead vs. Partition Size

While doing our preliminary experiments, we found that Trie property accounts for a good proportion of the system overhead, and, thus, is a good place to start our overhead analysis. We collected Trie overhead information for partition size varying between 0 and 100. The results are shown in Figure 4. It can be noted that the relationship appears linear, and so we ran a linear regression to determine the coefficients of the $\text{TrieOverhead}(\text{partition_size})$ function. The R^2 of the regression was over 0.99, indicating that this relationship can indeed be approximated by a linear function with regression coefficient of 9.79 and the intercept of 40.77.

We then experimentally evaluated the effect of varying trie partition size on access latency in the trace tests described above. The network speed was 10 Mb/s LAN connection to the server. The results are shown in Ta-

Partition Size (%)	Avg. Access Latency (ms)
0	961
10	876
20	1174
30	1331
40	1325
50	1664
60	1749
70	1668
80	1980
90	1797
100	1946

TABLE II

TRIE PARTITION SIZE VS. AVERAGE ACCESS LATENCY

ble II. This shows that optimal partition size is small for a high-bandwidth connection. The results would be more interesting for higher latencies, where the inherent network delay would mask whatever Trie overhead has been introduced. We plan to investigate this further in the future.

B. Apache Source Compilation

Bandwidth	Baseline	SpecialistsWinnow
10 Mb/s	270	260
1 Mb/s	683	677
500 Kb/s	930	914
100 Kb/s	1513	1496
56 Kb/s	3028	3023
28.8 Kb/s	7055	6996

TABLE III
TIME TO COMPILE APACHE SOURCE IN A
NETWORK-MOUNTED DIRECTORY (SECONDS)

To test our system with a more contemporary working set, we placed the source code tree of the Apache web server (approx. 22.5 MB) in a coda directory. We then ran test runs, starting from a cold 15 MB cache, and did “make clean” and “make” on this source. Our results for the unmodified Coda baseline and our system (SpecialistsWinnow) are shown in Table III, for the same range of bandwidths as for our trace replay tests. All tests were repeated three times and the average used.

The results show a small but significant reduction of user wait time for the entire range of bandwidths tested. One would expect our system to perform well for such a test, as source compilation of this sort generally will feature periods of network fetches, followed by periods of CPU activity while the files are compiled. During these compilation periods the network link would be relatively idle, and therefore prefetch requests could then be serviced without making any other requests wait in Coda’s queue.

V. CONCLUSION

File prefetching in distributed file systems has been a topic of research for many years, and a wealth of suggestions for predicting file accesses can be found in the literature. Our contribution is to provide a convenient framework to combine the numerous prefetching techniques into one powerful predictor, which automatically adjusts to its environment. While most studies of file system access patterns only report aggregate results (for the sake of statistical significance), different users

clearly will have somewhat different access patterns. This motivated us to pursue our composite approach.

Our results show up to a 15% reduction in file access latency on trace replay for the low-bandwidth, high-latency connections which are often a fact of life for mobile users. Our system also outperformed Kroger’s trie, a competing technique, on all tests. While these results are very encouraging, our system currently relies on several performance/overhead tradeoffs, which we manually set according to empirical evidence. Clearly, it would be preferable to have the system self-tune the relevant parameters, and this is a focus of ongoing work. We believe in particular that optimal, automatic tuning of the algorithmic learning parameters would result in improved performance for our Apache compilation tests, since we expected better results than were obtained.

It is clear our system is often of little use for low-latency connections, and in fact, imposes an overhead in those cases. Consequently, we are exploring the effects of making the system aware of the server connection status. It would seem ideal for our system to back-off and not waste computation time when it cannot help the situation, but then spring back into action when network conditions deteriorate.

It is possible that technological improvements (such as dramatic increases in Microdrive speed, energy efficiency and cost) could obviate the need for better file prefetching techniques. We expect in the short term, however, that the the lack of ubiquitous, high-speed wireless network access, combined with increasing working file set sizes, will continue to impact mobile file access performance. In the meantime, systems such as ours meet a need, by simply and cheaply enhancing mobile users’ computing experiences.

REFERENCES

- [1] Ismail Ari, Ahmed Amer, Ethan Miller, Scott Brandt, and Darrell Long. Who is more adaptive? ACME: Adaptive caching using multiple experts. In *Workshop on Distributed Data and Structures (WDAS 2002)*, March 2002.
- [2] Patrick Eaton Dennis. Clump: Improving file system performance through adaptive optimizations, December 1999.
- [3] Dennis Geels. Space-optimized markov chain model for file prefetching.
- [4] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching, 1995.
- [5] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer*, pages 197–207, 1994.
- [6] Terence P. Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for Web caches: Differential quality-of-service and aggregate user value. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

- [7] Thomas M. Kroegeer and Darrell D. E. Long. Predicting file-system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 319–328, 1996.
- [8] Tom M. Kroegeer and Darrell D. E. Long. The case for efficient file access pattern modeling. In *Workshop on Hot Topics in Operating Systems*, pages 14–19, 1999.
- [9] G. Kuenning. Design of the SEER predictive caching scheme. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, U.S., 1994.
- [10] Geoffrey H. Kuenning. SEER: PREDICTIVE FILE HOARDING FOR DISCONNECTED MOBILE OPERATION. Technical Report 970015, 20, 1997.
- [11] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. In *Symposium on Operating Systems Principles*, pages 264–275, 1997.
- [12] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.
- [13] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.
- [14] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 256–261, 1992.
- [15] T.M. Madhyastha and D. Reed. Intelligent, adaptive file system policy selection. In *Proc. of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, October 1996.
- [16] Brian Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Measurement and Modeling of Computer Systems*, pages 138–149, 1994.
- [17] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, 2001.
- [18] F. Rosenblatt. The perceptron: a probabilistic model for information storage and retrieval in the brain. *Psych. Review*, 65:386–408, 1958.
- [19] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.
- [20] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, 1997.
- [21] Vivekanand Vellanki and Ann Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, 1999. ACM Press and IEEE Computer Society Press.