File Prefetching using On-Line Learning

Zaher Andrawis, Anthony Nicholson, Yevgeniy Vorobeychik
Electrical Engineering and Computer Science
University of Michigan
{zandrawi, tonynich, yvorobey}@eecs.umich.edu

Abstract

Distributed File Systems suffer from a well-known network latency bottleneck, and caching has long been used to alleviate this problem. Indeed, many researchers have proposed enhancements to the basic cache-on-demand and LRU mechanisms, suggesting various ways future file access patterns can be predicted and files prefetched into the local cache accordingly. Our insight is that we can do better by combining multiple predictors, or properties*, into one global mechanism that listens to all suggestions and combines them into one final prediction. We do this by allowing our system to learn how each property performs over time and then splitting the cache between the properties based on performance. When the environment changes, our algorithm dynamically adjusts, or "re-learns". Our resulting model is simple and robust, as it supports an arbitrary number of properties that can be easily integrated with the rest of the system as long as they export a predefined interface consisting of only three functions. We have implemented the system proposed here within the Coda distributed file system and ran a series of benchmarks. Our results indicate up to 39 percent improvement in average access latency as compared to the baseline implemenation for low-bandwidth connections.*

1 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. However, as the average workstation becomes increasingly powerful, the significance of these bottlenecks only seems to increase. Even though we have entered the era of Gigabit Ethernet, network latency is still an issue, and, generally, once the client leaves the server's LAN, the problem very quickly becomes critical. Thus, improving the performance of caching can dramatically enhance overall performance. Since we are looking to improve performance over high-latency network connections, we concentrate on mobile devices with limited local storage capacity. If the storage capacity is large, our system will not be of much help, since working sets tend to be relatively small and may, in fact, fit entirely in the local cache.

Cache performance improvement can be approached in two ways: one may 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 (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].

One of the major strengths of the system we developed 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.

The actual prefetching is performed by the prefetcher module, which interfaces directly with the Coda cache manager, Venus. Our system is currently implemented inside Venus, though the prefetcher is the only component that actually relies on it. The algorithm module relies only on the interface provided by the prefetcher and does not interact with Coda code. This allows easy portability of our system to other environments[1]. Again, such portability was an important design goal, and we believe that we have succeeded in achieving it.

Finally, the flexibility of our system is one of its most unique aspects. It can be expected to perform well in different environments, as different properties may be designed to work well in very specific settings, but collectively will work in the union of these. This is expected due to the convergence property of our algorithm: it will dynamically redistribute the weights to assign a relatively high proportion of the cache to the properties that are successful *in the current environment.*

The Related Work section which follows spends some more time discussing the various approaches to file prefetching that we have encountered in literature. Section 3 delves deeply into the design and implementation aspects of our system, including a description of the properties we have implemented. This will include a detailed discussion of the internal workings of the Algorithm Module (Section 3.3), and the description of how all the components in the system fit together. To test our system, we have done a performance evaluation using Coda Traces[2] to drive the file access simulations. Additionally, we tried to evaluate the performance of our system in a development environment by running a make of Apache server. Our simulations and results are described in detail in Section 4.

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

[2]These traces were collected at Carnegie Mellon University in 1991-1993

2 Related Work

One of the early 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*[15, 18]. 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. In its later development phase, SEER also incorporated directory membership, "hot links", and file naming conventions into the hoarding decision process.

In 1994 Appleton and Griffioen published their work on prefetching[4, 5]. Their approach 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 afterwards. The weight of each arc is the number of times it has been visited (i.e.,the number of times the second file was accessed within the lookahead period of the first). Thus, if some file is 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.

The next important work in the field was by

Kroeger[7, 8]. His work included 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 reminicent of Appleton and Griffioen's model, the children of a 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 tried to improve on those discussed above. The CLUMP project[2] attempts to use 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 Chervenak revisted the Patterson's Cost-Benefit analysis, but adapted it to a probabilistic prefetching environment[19]. 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 our that we have found in literature. The main difference is that we concentrate on prefetching, whereas this algorithm deals with replacement strategies.

Still, 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, probability graph per Appleton's work with several other properties that we feel are 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 different environments. Our solution allows a set of predictors to dominate the prefetching decisions in the environments to which they are best suited, and to give way to others when those become more successful.

3 Design and Implementation

3.1 Design Overview

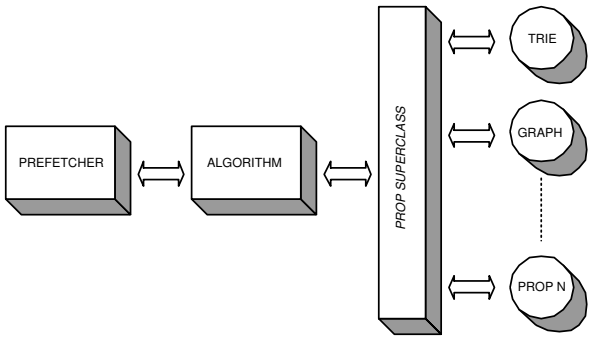

Figure 1: System Design

Quite unsurprisingly, we believe that network file prefetching should be done within a Distributed File System. We chose Coda [17] 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 wealth of file traces and examples of valid measurement techniques [14]. 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 one can see from Figure 1, we have broken our logic into several parts. The primary motivation for this was portability. Only the Prefetcher module contains OS-specific code, and would therefore be the only code to change in order to port our system to new operating systems.

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 activ-

ity, 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 datastructures. We have defined `class algorithm` as a singleton; as it is only accessed from the File System thread. However, exclusive access control on the object is not a concern.

3.2 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 is a major plus.

Venus is comprised of many threads, but has only one file system thread. Fortunately for us, this thread handles all the functions described above, so we do not have the headache of thread coordination and concurrency. 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 the latter 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 as 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, avoiding an endless loop.

3.3 Algorithm Module

3.3.1 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 *synchronously*[3]. In turn, the algorithm itself relies on a set of *properties* to make their local predictions, which it then manipulates to come up with a global set of predicted files. While all properties are created equal, they diverge in importance, as some may be observed by the global decision-making unit to be "weak" predictors. Thus, the algorithm is analogous to a president delegating decisions down to advisors, with some advisors 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 have 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?

[3]This is important because some properties (for example, the trie) need to know the actual sequence of file accesses.

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 trying setting our choices in stone, we created a modular architecture which would allow any additional predictor to be simply "plugged-in" into the algorithm. This allowed us to concentrate on evaluating our model using only a small set of properties, which we hypothesize will produce the best results. We chose to use the Trie[7, 8], Last Successor[8], Probability Graph Successor[4, 5], Directory Membership, and File Type (extension) as our initial properties, which are described in greater detail in the following sections.

Each property exports the following simple interface to the algorithm:

- *file_accessed(file)*: notify the property of a file access event
- *get_prefetched_list(size)*: ask for the list of predicted files that fits into size
- *confidence(size)*: ask for the confidence of the property in the currently suggested list of predicted files (sum of probabilities); if not provided, confidence of 1 will always be assumed

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 $p = 0,1,...$ is the index.

3.3.2 On-line Learning in the Algorithm Module

So how does this black box reconcile all the information contained in the properties? The answer, of course, is online learning. Our learning approach is similar to the Rosenblatt algorithm, though we have revisited the process which makes the final decision. The Rosenblatt algorithm [16] reacts to counter-examples (misses) by adding the current input to the weights; otherwise, the weights remain fixed. We used this algorithm when the input is the ranking of the files by each property. During a cache miss, each property predictor is asked to supply its ranking. This policy leads to boosting the weights of properties that considered the missed file more important.

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 overall ranking as $\sum_{p \in properties} w_p \cdot r_p$ 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 more fair towards 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 *common files* that will inevitably be 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 is stable. Furthermore, the property/algorithm interface is greatly simplified, facili-

tating 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 a proportion of the cache equivalent to its weight divided by the sum of all weights.

The more difficult question of determining the weights for all properties is answered as follows:

```
for (each property) {
    if (cache miss) {
        //emulate retrieving prefetch list with property thinking
        //that it has the entire cache
        //This is done for the file ranking information to be
        //stored internally by the property class that will be
        //returned by the susequent notify_file_accessed call
        //note: the list of files returned here is never used
        property→get_prefetch_list(entire_cache)
    }
    //notify each property of the access
    //receive the ranking of the file
    //according to its placement
    //in the prefetch list previously
    //retrieved (get_prefetch_list call above)
    rank_of_file = property→notify_file_accessed(file)
    if (cache miss) {
        //update weights according to the rankings retrieved
        weight[property] = weight[property]+rank_of_file
    }
}
```

Figure 2: Pseudocode for computing weights

As can be seen from the 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 ranking=$\frac{1}{1+p}$, which is just an arbitrary decreasing function in p

When the algorithm module receives file lists from all the properties, it has to merge them in order to send one complete list back to the perfetcher. This merge generally needs multiple iterations of the get_prefetch_list queries, since existence of files that are common to several properties frees space for additional suggestions.

3.3.3 Boosters

While our learning mechanism is very flexible, it does not account for short-time variations in the success of the properties, concentrating, instead, on the long term stability. Many environments, however, may be inherently unstable (context switching, for example, can confuse the predictors), and to account for this we decided to create a set of *boosters* within the Algorithm Module logic. We have currently implemented two boosters described below, but our Future Work section provides descriptions of several others that could improve the performance or usefulness of our system.

CI: Confidence Indicator Although weights are a good indicator of the level of success of each property, at different points in time different properties may have varying "confidence" about their predictions. To account for this, we added a method to the property interface that returns a number between 0 and 1 indicating how confident the property is about its current predictions. Each time the algorithm is asked for a list of files to prefetch, it multiplies each weight by the confidence indicator of the respective property and then goes on to do the usual operations to put together the predicted file list it will send back.

In designing CI, we had to make sure its definition is globally meaningful and fair to all properties. We decided that cumulative probability of files that fit into the cache slice allocated to a property satisfies both of these constraints: all properties sort predicted files by probability of these files to be accessed in the future, and fairness is achieved in the long term since if CI reduces cache slice of a "good" property, the corresponding increase in the weight over time will eventually compensate for this.

PRO: PRoperty Observer Weight alone does not indicate the absolute usefulness of a property –

a property may be ineffective when it suggests files that are already suggested by other properties. This module simply checks which property uniquely predicts the currently accessed file. If no such property is found, the aging indicator of all is incremented. If, however, such property is found, its aging indicator is cleared. If the aging indicator of any property exceeds some threshold, this property becomes passive: it is no longer used by the algorithm to compile its list of predicted files, though it still learns of all file accesses and its weight is still being maintained according to the previously described scheme. If at any point a property that is passive uniquely predicts any file, it is immediately activated. To make sure that similar properties do not become passive simultaneously, PRO only makes one property passive during each file access.

3.3.4 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.

Trie The trie property was created using a multi-order context model as 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. This can be easily seen in a development environment, where a make file 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-Complete[4], and, therefore, we ended up using 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 advice[8] and implemented constant partitions. Indeed, our results show that varying partition size had a significant affect on the trie overhead.

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 as 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 similar to the semantic distance used in SEER.

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.

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

[4]It can be reduced from the Knapsack problem.

based on directory distance. Distance of 0 between two given files indicates that they are in the same directory, distance of 1 means one file is the direct descendant of the other, and so on. Predictions are made by following the directory hierarchy and ranking files according to directory distance.

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.

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.

File Extension Intuitively, this again seems like a good indicator of access locality, as it is easy to envision users opening several StarOffice or pdf documents within a relatively short time, particularly since it takes extra effort and time to open and close such applications, and few like to spend more effort than the minimum necessary.

File Extension property works by saving the directory of the last file accessed and records extension sequences (shifts) similar to the Probability Graph property. It creates its prediction list by scanning the directories in the order of increasing directory distance metric as described earlier, returning only the files with the extensions corresponding to the successor database.

4 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. We used a Linux kernel module (NistNet) to simulate various network latencies, although our test server and client were in fact directly connected via an Ethernet crossover cable. Both test machines ran Linux kernel 2.4.18-3 (Red Hat Valhalla), 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 explore. First, (as described below), we obtained traces of actual file access on a server at Carnegie Mellon University. We replay these traces to compare the performance of our modified system to baseline Coda, to a Trie approach (Kroeger's solution), and combinations thereof.

As stated below, these traces are not the most current, so we also defined two additional tests. We placed the source tree of the Apache web server in a Coda-shared directory, and measure the compile time for our system, baseline, and Trie. JSince these tests involve transferring a great deal of data, we expect see a high miss rate for the baseline system, and are interested to see how much our system will mitigate this.

4.1 Evaluations using Coda Traces

The following evaluations were performed with file access calls simulated from the Coda Traces taken from the mozart computer at CMU in 1993 for a period of about one month. This particular machine was chosen because it is described as being a "typical" workstation. While the Coda Traces are almost eleven years old, they were the most convenient for us to use, partly due to our preliminary experience with them. Our goal for the future is to involve traces that are considerably more recent and rerun the same tests.

A few words need to be said about how these tests were set up. 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 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. In the future we intend to

improve this by selecting the sizes for zero-byte files (files for which the size information is missing) from a realistic distribution reported by a recent file system study.

4.1.1 Network Latency Evaluation

We hypothesized our system would be more beneficial as network latency to the file server increased. As network delay increases, whatever computational overhead our system has introduced should be overshadowed by the network delay to fetch a file. Since our system should reduce the overall number of fetch-on-demand operations, this is of interest, for such bandwidth limited devices as wireless PDAs.

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

- no properties active (baseline)
- Trie only
- all properties (our full system)
- all properties + boosters
- DirLRU property only
- Trie + Dir property + DirLRU property
- Trie + Dir property + DirExtension property

We added the last three tests out of curiosity. If a subset of our properties performed much better than the whole suite, it would lead us to believe we need to adjust more quickly to those properties which are performing well and those which are not.

For each test listed above, we used the network latency simulator to replicate roundtrip time (RTT) to the file server of 1ms, 10ms, 50ms, and 100ms. We experimentally verified RTT of 1-10ms is comparable to a LAN connection, 50ms to that of a cable modem / DSL link, and 100 ms and above are typical for wireless links or dial-up modem connections.

Table 1 shows the average access latency vs. RTT, for each prefetcher configuration. Average access latency is calculated as the total time required to replay the Coda traces divided by the total number of file accesses.

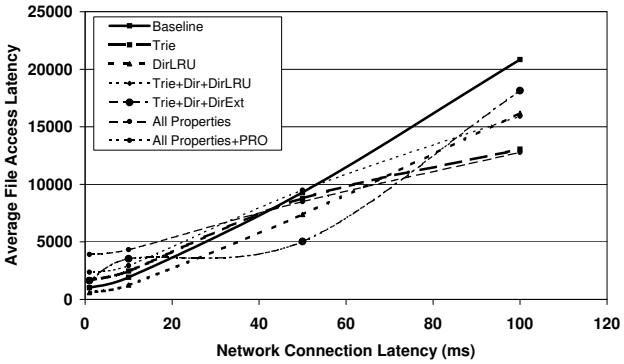

Figure 3: Avg. Access Latency (ms) vs. RTT

Prop	1 ms	10 ms	50 ms	100 ms
none	1010	1923	9296	20844
Trie only	1684	2455	8784	13059
DirLRU only	598	1248	7389	16188
Trie+Dir+DirLRU	1630	3529	5027	18147
Trie+Dir+DirExt	1530	2930	6717	15442
all	3910	4318	8494	12785
all+PRO booster	2362	2940	9503	15922

Table 1: Average Access Latency (ms) for Coda traces vs. RTT to Server, per Property Configuration

The results show our system starts at approximately 3X the baseline access latency for LAN speeds but as the RTT increases, it begins to out-perform the base Coda implementation. One can conclude that on the small RTT runs, our poor performance is due to our inherent system overhead. As file access latencies increase for higher RTTs, this overhead becomes less important, and the fact that we are prefetching files causes enough cache hits to make the difference. At the high-end of the RTT spectrum (100ms), we show a 39% improvement in access latency over the baseline for the 100ms RTT case. This corresponds to our target application (weakly-connected mobile devices).

In comparision to an optimized Trie, we perform quite well. Our avg. access latency is nearly identical for the higher-RTT cases (50ms, 100ms). We expected this, as our system is a prototype, and

the Trie we used is optimized, benefiting from the work already done by Kroeger and others. One can see that 3 subset runs (DirLRU, Trie+Dir+DirLRU, Trie+Dir+DirExt) actually beat the Trie for the 50 ms case. This suggests that our overall performance (running with "all" properties) could be improved if we can improve the process of assigning weights, to ensure the best properties of the moment are contributing fully, since these properties are more accurate than the Trie in those cases.

An interesting side-note is the performance of the PRO booster. It seems to improve performance in low-latency cases but in the longer term does worse than our regular "all properties" configuration. It is possible it may be ejecting properties prematurely, that might be useful. This is an optimization we will pursue.

4.1.2 Trie Overhead

Our ultimate goal is to make this system adaptive to network speeds, 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 end-user utility function, which we assume to be an inverse function of average file access latency per byte.

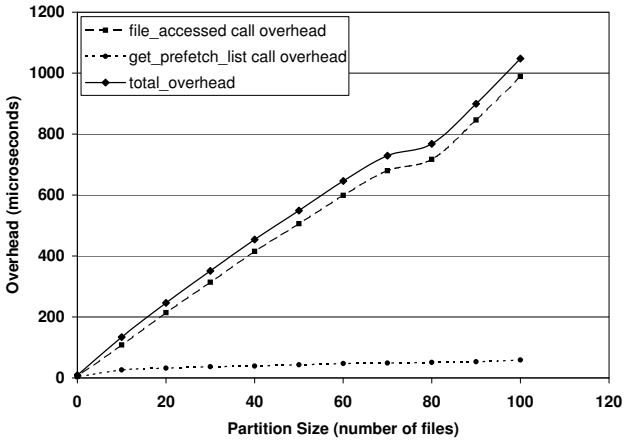

Figure 4: Trie Overhead vs. Partition Size

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, so we ran a linear regression to determine the coefficients of the TrieOverhead(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.

Partition Size	Avg. Access Latency (ms)
-1	1151
0	961
10	876
20	1174
30	1331
40	1325
50	1664
60	1749
70	1668
80	1980
90	1797
100	1946

Table 2: Trie Partition Size vs. Average Access Latency

We then experimentally evaluated the effect of varying trie partition size on access latency in the trace tests described above. The network speed was 100 MBit LAN connection to the server. The results are shown in Table 2. This shows optimal partition size is small for a high-bandwidth connection. The results would be more interesting for higher latencies, where Trie overhead does not play as much of a role, and we plan to investigate this in the future.

4.1.3 Apache Source Compilation

For an additional, real-life test, we placed the source code tree of the Apache web server in a coda directory. We then ran three test runs, starting from cold cache, and did "make clean" and "make" on this source. Our results for baseline, Trie only and all properties are shown in Table 3.

Prop Set	Build Time (ms)
all	550
trie	561
none	438

Table 3: Apache Build Time for each property set (ms), 5 ms RTT

Our system performs better than the Trie alone but still slower than the baseline. We believe this is due to our inherent overhead which becomes visible for fast connections (5ms RTT in this case).

5 Future Work

5.1 Reducing the Magic

Our system right now relies on several parameters which we manually set according to our intuition about the trade-offs of performance vs. overhead. We believe that this is the wrong way to do it and simply did not have time to implement these away. Time limitation is not our only excuse for keeping around many floating parameters tbat we tweak in the configuration file. We have to do this first in order to collect data on how these parameters vary the effectivenes of our prefetcher, and using that data (after a few regressions) to create functions we can later use in the optimization algorithms that set these dynamically.

One of the most important parameters we set is the cache slice (percentage) alloted to our system. The algorithm for setting it dynamically will maximize a "worthiness" function, which is defined as $w(size, speed_{LAN}) = (T_{LAN} - T_{hit}) \cdot (hit\_ratio\_prefetcher(size) - hit\_ratio\_lru) - T_{overhead}$. Note the similarity between this and Patterson's cost-benefit analysis[15, 18].

In addition to the cache slice parameter, trie partition size and threshold can be dynamically set by the system depending on the current connection speed. Again, this can be done by defining some utility function as U(partition_size) = benefit - cost, with benefit being an inverse function of average latency, and cost being the prefetching overhead and execution overhead. For execution overhead estimation, we will use the regression results from Section 4.1.2 to create the TrieOverhead function:

5.2 Enhancing the Algorithm with Monitor Modules

To further enhance the performance of our algorithm, we came up with several ideas for modules we called *monitors*. These are described below.

STAD: STAbility Detector Keeps track of how stable the weights of the properties are by comparing the sum of absolute variations of all weights to some threshold and updating the status in the algorithm module accordingly.

KNOWR: KNOWledge Recorder This monitor simply dumps all the learning database information to file either for backup or to use on different workstations or for different users.

HUMAD: HUman versus MAchine Detector The file access behavior of a human versus an application often affects the file access pattern, and this monitor would try to take advantage of this additional information by measuring time between consecutive file accesses for some number of files, and if this time is "too short" (i.e. below some threshold), the "machine mode" is assumed by the algorithm. Conversely, if the time between consecutive accesses is "long", it must be a human opening the files. This information may be useful in determining the set of properties that the algorithm favors when working in a human or machine mode.

SICO: SIze COnsultant This monitor module relies on feedback from the prefetcher module about the proportion of files last suggested by the algorithm that was actually prefetched (the rest ignored since they already were in the cache). The algorithm may use this information to decide that it needs to increase the size of the predicted file list it returns to the prefetcher in order to compensate for this "waste".

Conversely, algorithm may decide that it is using too high a percentage of total cache and evicting some important files. An additional subtle usefulness of this monitor is to implement that actual fraction of the cache that the algorithm uses to prefetch files, as long as we are assuming that our algorithm module is not going to maliciously oversaturate the cache (if this assumption doesn't hold, we may be in trouble anyway).

6 Summary

Considering our system is an initial prototype, it appears to have performed very well in this evaluation, showing a 39% improvement in access latency over the baseline for the 100ms RTT case. This corresponds to our target application (weakly-connected mobile devices). It is reasonable to expect performance can be improved to approach baseline for strongly-connected cases, as our currently poor performance is due to inherent computational overhead. Furthermore, our results are as good or better than a Trie alone for high RTT, which leads us to conclude our composite approach is a promising one, and worth of further study.

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. Kroeger 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. Kroeger 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] 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*, 1996.

[14] 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.

[15] 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.

[16] F. Rosenblatt. The perceptron: a probablistic model for information storage and retrieval in the brain. *Psych. Review*, 65:386–408, 1958.

[17] 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.

[18] 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.

[19] 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.