Analysis
Clustering algorithms
for pre-fetching
in Motefs

Narendar Shankar Vijay Gopalakrishnan

{narendar,gvijay}@cs.umd.edu

Department of Computer Science,
University of Maryland, College Park.
May, 2001.

Introduction.

Most of today's computing is highly dependent on networks. The resources and files are shared over the network. There are servers to which the users connect to and access their valuable data. Motefs is such a system that aims at providing ubiquitous access to files, at minimal cost. Ubiquitous connectivity is becoming more of a reality, but ubiquitous access to files by the file system has not matured completely. Of course file systems like NFS [1] and AFS [2] have not been popular on the wide area network because of administrative and security issues. Systems like Oceanstore [3] have a very complex model for data storage and location, which might have severe performance hits. Motefs on the other hand uses a combination of hoarding and pre-fetching with some intelligent analysis of both file reference patterns and network connections.

Hoarding has primarily been identified as a solution for disconnected mode operations. In the absence of proper network connections the user is forced to work in a disconnected mode or resort to the worst case - stop all work. The solution to this is to "hoard" the files into the client computer. Hoarding eases disconnected operation by selecting a subset of the user's files for local storage. A hoarding system works by observing user activity and predicting future needs. Thus a hoard database is built and all hoarded files are downloaded onto the client machine.

The other alternative is pre-fetching of files, which is used there is a network connection. A critical architectural feature of such systems is the caching of data at clients. It has been recognized that client side caching can be exploited to address the performance and availability issues. In particular, pre-fetching increases cache hit rate and reduces file read latency.

Motefs uses both these techniques effectively. Hoarding databases are built on the server side and on the client side pre-fetching is done based on the files in the hoard database in the server. Further, Motefs uses a new clustering algorithm, which creates clusters based on network conditions in a dynamic fashion. We believe that such dynamic clustering has not been looked into before, but nonetheless is very important because of fluctuation network conditions.

Related Work

Hoarding of files had been pioneered by the Coda [4] file system, which used Venus (the cache manager of coda) for hoarding files. After that the SEER [5] system had tried to build hoard databases based on semantic information between files. We simulate their method in our experiments. There has been more recent work on using access trees [6] that focuses on better semantic relationships between files. We are looking into these methods currently. Our work primarily focuses on network-conditions-adaptive clustering and also on semantic relationships between files and this can greatly benefit from better semantic clustering algorithms.

Design and Implementation

File trace collection

We built a system call logger in Linux, which traps system calls related to the file system and logs them. We also use a script that generates file access patterns for multiple users. Once multiple users have run the script, we separate the log into per user logs. We also ensure that the logs are more or less stable by rerunning the scripts till the difference between the logs is minimal. We use the standard Unix diff for this purpose. We use this log to generate clusters and also use it for the reference stream.

Clustering algorithms.

The difficult challenge is the ``hoard database building problem''-i.e. of selecting which files should be stored locally. The simplest way is to ask the user, which files to hoard. But this method involves the expertise and involvement of the user. A more elegant solution would involve the hoarding of files without involving the user directly
Automated predictive hoarding is based on the idea that a system can observe user behavior, make inferences about the semantic relationships between files, and use those inferences to aid the user. We have looked into three different types of algorithms that could automate it.

a. Temporal clustering

Temporal clustering clusters files together based on timestamps of access. We however believe that it is incorrect because it depends on user's access times, which varies between users and also between different sessions for the same user. Though we have it incorporated as a part of our simulator, we are not presenting results for these because of correctness issues.

b. Semantic clustering- the SEER system.

The SEER system developed by Geoffrey H. Kuenning and Gerald Popek calculates a new measure, semantic distance, between individual files, and uses this to feed a clustering algorithm that chooses which files should be hoarded.

SEER considers the user's activities to be composed of projects, rather than individual files, which greatly enhances the accuracy of its predictions. The system watches the user's file access pattern and records each access according to whether it is an open or a close. The file reference patterns are evaluated and are used for calculating the semantic distances among various files. These semantic distances drive a clustering algorithm that assigns each file to one or more cluster.
To detect semantic locality,

SEER defines a new concept known as semantic distance. Conceptually, semantic distance attempts to quantify a user's intuition about the relationship between files. A low semantic distance suggests that the files are closely related and thus are probably involved in the same project, while a large value indicates relative independence and different projects. Semantic distance is based on measurements of individual file

references, rather than looking at the files themselves. In the system, a file reference is considered to be a high-level operation, such as an open or status inquiry because doing so would obscure the information being extracted. The idea is to get whole files, rather than individual bytes, so it is more informative to look at whole-file operations.

The lifetime semantic distance between an open of file A and an open of file B is defined as 0 if A has not been closed before B is opened, and the number of intervening file opens (including the open of B) otherwise. For example, consider the reference sequence {Ao, Bo, Bc, Co, Cc, Ac, Do, Dc}, where the letters o and c indicate opens and closes respectively. The lifetime-based semantic distance from Ao to each of Bo and Co will be 0, while the distance from Ao to Do will be 3. Similarly, the distances Bo to Co, Bo to Do, and Co to Do will be 1, 2, and 1, respectively. All other distances (Bo to Ao, Co to Ao, Do to Ao, Co to Bo, Do to Bo, and Do to Co) are undefined.

Once we have generated the file reference pattern of the user, we are ready to calculate the semantic distance. But now comes the problem that there could be multiple references to files when the distance is compared with on file. There are two ways to handle it
1.) Perhaps the simple way to do it is to use arithmetic mean of the distances. But this suffers from the problem that if there are many distances that are small values, but one that has a very big value, this big value affects the distance. The problem is that small number is more indicative of a relationship.
2.) Use geometric mean, which gives smaller, values more importance.
So we use either of the methods to find the semantic distance of each file w.r.t the other files. This is now used to feed the clustering algorithm and generate clusters.

The algorithm developed is a modified version of one originated by Jarvis and Patrick. This algorithm is bottom-up, or agglomerative, starting with each data point assigned to an individual cluster and then combining clusters according to a shared-neighbors criterion. In this algorithm, the 'n' nearest distances to a file is got from the semantic distance calculation. There are two thresholds, Kn(Knear) and Kf(Kfar), Kn > Kf. If two files share at least Kn neighbors, then their clusters are combined into one. If the two files share fewer than Kn but at least Kf neighbors, their clusters are not combined, but instead are overlapped. In the overlapping operation, each of the closely related files is added to the other file's containing cluster. Files, which share less than Kf, are left as is in their clusters.

Relationship	Action
Kn <= x	Clusters combined into one
Kf <= x < Kn	Files inserted, but clusters not combined
x < Kf	No action

Table 1. Deciding when to clustering files.

c. Bandwidth adaptive clustering

We believe that for a ubiquitous file system has to base its clustering algorithms on both user access patters and also on bandwidth. The network bandwidth is an important factor because when entire clusters are being transferred over the network large clusters take a

long time to arrive and more importantly might cause the client cache to be filled up and possibly be even overwritten by the same cluster(if a cluster is bigger than the cache). In bandwidth adaptive clustering, we initially form clusters based on weights for ordering. In the reference stream used to build clusters, for each file we have a weight vector for "cluster-size" of nearest files. In other words we give weights in descending order for files, which occur near this file. The weight is based on the overall nearness of the file throughout the entire reference pattern. This is pretty similar to the first step in the SEER method.

In the next phase we change weights based on size and bandwidth considerations. The initial transfer time for a cluster is known and the client based on its perception of bandwidth can specify the desired transfer time. When we mean perception of bandwidth it is the client's instantaneous bandwidth calculated by measuring the roundtrip time and the amount of data transferred (b/w= amount of data/(RTT/2)).

But the above measurement is prone to error, so the client can use a much better long term estimate of bandwidth using a cheap application layer technique where the bandwidth is calculated by measuring application layer latency by using packet probes and resending the packets until a confidence bound of 95% can be established. We found that it was not necessary to send more than 100 packets to get a good confidence bound.

Based on the desired transfer time the clusters is recomputed so that the total size of clusters is reduced or the clusters are merged together if the desired transfer time is greater than the current transfer time. In our current implementation we only support the former, but we are looking at the other option also.

Experiments

We conducted our experiments in two phases. We ran a simulator where we could specify both the cache size and also the bandwidth. We also implemented a user level client and server and estimated bandwidth over the real internet. For the bandwidth adaptive clustering method, there were in-band signals sent to the server for re-clustering. Right now the in-band clustering is also a user specified parameter.

Simulating the transfer of files.

Once we generate the cluster, we have to use it to send files according to need. In order to simulate a request, we observed the access pattern of the user and logged it. This log was replayed to hoard the files. There was a specific amount of cache that was allocated. As the request came, we checked if the file was already in the cache. If not, a request was sent for the file. This request was satisfied in the form of the sending the entire cluster over, in the anticipation that the user will reference some other file of the same cluster. We noted the number of misses to the cache and the time to transfer the files over the network. During the calculation, we work under the assumption that the entire cluster is transferred before the file is available. So the order is unimportant.

Our experience with the real internet

We simulated the above behavior with a restricted cache size and also with an unrestricted cache size and observed the times taken for our clustering algorithm based on bandwidth adaptive clustering. Like mentioned before we have used both the long-term bandwidth and instantaneous bandwidth calculation for re-clustering. The long-term bandwidth though calculated at the application level is done before the client starts fetching files. In the future we plan to use a model in between the current models, where the bandwidth estimate is based on a model, which uses a few instantaneous samples rather than just one. A map of the internet with the four sites we had chosen look as follows.

Also because of memory restrictions, we could not transfer the entire file, so we decided to just transfer metadata.

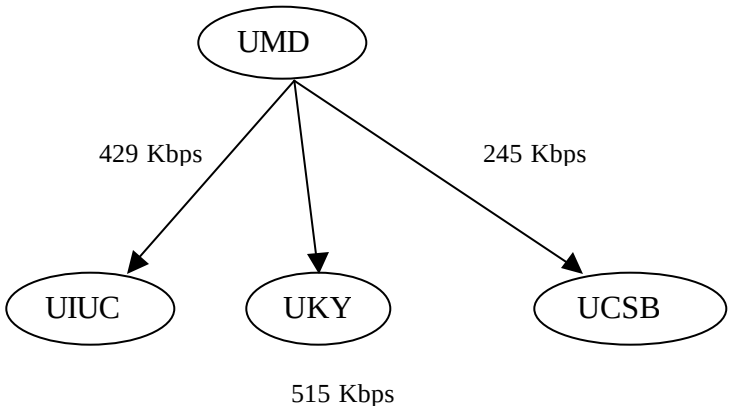

Figure 1. Map of the internet experiments

Test Results

We had a reference pattern for clustering of approximately 2500 files(397 unique ones). We also used a reference pattern of approximately 2000 files for the client references.

Results for SEER

When fed to this algorithm, with n set to 9 nearest files, Kn as 5 and Kf as 3, we got clusters 8 clusters. When n was set to 20 and Kn to 10 and Kf to 5 respectively ., we got just 2 clusters. We decided to carry on with 8 clusters because the time to hoard 200 files may be enormous.

Test case	512 K	1 M	2 M	4 M	8 M	16 M	32 M
Transfer with UIUC	4072s	7814s	13013s	21527s	18553s	378s	378s
Transfer with UKY	3532s	6548s	10740s	17901s	15417s	314s	314s

Table 2: Results of SEER adaptive clustering algorithm.

Observations.

- The miss rate decreased as the size of the cache decreased.
- The transfer time increased as the size of the cache increased up to a certain value and then decreased. This may seem contrary to what one would expect. but the reason for this is that some of the files in the access pattern were so large that they could not fit into the remaining space of the cache. In our simulation, files that could not be stored in the cache are not sent across. So that accounts for increase in time as the size increases. But after sometime the time decreases because, the cache is able to hold the files instead of throwing the files to replace it with newer files. So the number of bytes sent across decreases.
- Very few clusters, with many files are generated.
- Cache overflow was a problem.
- With 32 MB cache we got the same performance as unlimited cache.
- The threshold at which the times reduce is different for each input set. For eg. In this run, the results show just 314 sec. We believe that this is because the size of files is small, and since the cache is bigger, the files are transmitted only once, but in other instances, the files keep getting trashed. Due to this there are a lot of transfers.

Results for our algorithm

We conducted experiments for our algorithm in two different manners. First we simulate it using estimated bandwidths and compare it with the SEER method and show its effectiveness for varying cache sizes. We also show that the method stabilizes at a pretty low cache size on the client side(stabilization means that the cache size performance is the same as an infinite cache). We then show the effectiveness of adaptive clustering as compared with our normal clustering algorithm. We then show how the same results vary when instantaneous bandwidth calculation is used on the real internet. In the online algorithm re-clustering is done based on a percentage reduction, which is desired. By a percentage reduction we mean that the smaller files in a cluster are given higher weight by the percentage amount and then re-clustering is done so that some smaller files are also included in the cluster, which reduces the cluster size and hence the transfer time. We also reduce the size of each cluster by K files (after all weights have been normalized, k can again be specified.). In our simulations we entrust the intelligence of making a re-clustering decision to the user who specifies as a part of his feed file the desired percentage reduction. The results given below are shown for the *worst-case* behavior of the normalized algorithm with respect to the SEER algorithm where the access pattern causes severe trashing . Besides many of the big files have been removed from the reference patterns to favor the SEER algorithm. This favors the SEER method, which builds bigger clusters.

As can be seen from the results in table 3, the initial time taken for the transfer is low (because of the compulsory misses). But the access times stabilize at around a 1M cache (note that again here the we have many small files). Though the performance of this

method when compared with the SEER method is not very big (purely because of the nature of tests chosen to favor seer), it still does better than the SEER algorithm.

Method	64k	128k	256k	512k	1M	2M	4M	8M
Normalized clustering for UIUC	6417s	8399s	8399s	8321s	8314s	8314s	8314s	8314s
Bandwidth Normalized(5%) for UIUC	-	8219s	8247s	8267s	8214s	8214s	8214s	8214s

Table 3. Times for entire network transfer (in seconds).

Bandwidth normalized clustering performs better than simple clustering because of the fact that the size of the cluster gets reduced. Again here we have chosen a very small percentage for adaptiveness.

Percentage	5%	10%	20%	25%
Value for 1M cache	8214s	8211s	8201s	8268s

Table 2. percentage

We have found out that there is a threshold point of percentage decrease in size, which is beneficial, and this is around the 20% mark. However it must be noted that percentage reduction in our internet experiments are also user specified values and the online mechanism uses specific targets in the feed file to recognize it.

Conclusions and future work

This paper talked about a bandwidth adaptive scheme for a file pre-fetching model. Previous work laid importance on hoarding and clustering algorithms, which form one side of the story. We really believe that that bandwidth adaptiveness or in general adaptiveness to network characteristics are very important for ubiquitous file systems of the future. We are investigating further areas like adaptiveness to other network conditions and also at some better clustering algorithms, which use context sensitive information for identifying clusters.

References

[1] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., "*Design and Implementation of the Sun Network Filesystem*", Proceedings of the Summer 1985 USENIX Conference, Portland, OR, June 1985, 119-130.

[2] L. B. Huston and P. Honeyman. *Disconnected Operation for AFS*. In Proceedings USENIX Symposium on Mobile and Location-Independent Computing, pages 1--10, Cambridge, Massachusetts, August 1993.

[3] J.Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. *Oceanstore: an architecture for global-scale persistent storage*. In ASPLOS 2000, pages 190-201, November 2000.

[4] J. J. Kistler and M. Satyanarayanan. *Disconnected Operation in the Coda File System*. ACM Transactions on Computer Systems, 10(1):3, February 1992.

[5] Geoffrey H. Kuenning. *The Design of the Seer Predictive Caching System*. In proceedings of Mobile Computing Systems and Applications 1994, Santa Cruz, CA, December 1994.

[6] Hui Lei and Dan Duchamp. *An Analytical Approach to File Prefetching*. In Proceedings of the USENIX 1997 Annual Technical Conference, pages 275--288, January 1997.