IN-63-TM

47976

Predictive Caching using the TDAG Algorithm

Philip Laird
AI Research Branch
Ronald Saul
RECOM Technologies
Artificial Intelligence Research Branch
MS 269-2
NASA Ames Research Center
Moffett Field, CA 94035-1000

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-92-30

December, 1992

Predictive Caching using the TDAG Algorithm
(Technical Report FIA-92-30)

Philip Laird Ronald Saul

AI Research Branch
NASA Ames Research Center
Moffett Field, California 94035 (U.S.A.)

Abstract

We describe how the TDAG algorithm for learning to predict symbol sequences can be used to design a predictive cache store. A model of a two-level mass storage system is developed and used to calculate the performance of the cache under various conditions. Experimental simulations provide good confirmation of the model.

Introduction

A *cache* is a multi-tier store consisting of a hierarchy of storage media with different access speeds, capacities, and costs. The medium with the highest capacity is simultaneously the one with the least cost per byte and the slowest data access rate. In a typical mass storage system (MSS), for example, the majority of archival data is kept on tape, while the portions in active use are retained on disk or in RAM.

A request for a segment of data that is not available in the higher-speed store (or *cache*) is answered by replacing one of the current segments in the cache by the requested segment. Moreover, if the segment being replaced has been modified, it must be rewritten to the slower storage medium. In view of the penalty in waiting time to be paid whenever a requested segment is not in higher-speed store, the choice of which segment to replace can have a significant impact on the performance of the system.

A variety of algorithms have been used in an effort to minimize transfers of segments between the cache and the secondary store. The LRU (least-recently-used) strategy, for example, replaces the segment least recently referred to by the user(s), on the assumption that the more time since a segment was requested, the less likely it is to be requested soon. The general idea of LRU and other such strategies is, of course, to predict future segment usage.

The cost of making such a prediction, however, cannot be ignored. In virtual storage systems, for example, only extremely fast prediction algorithms can be considered. In other

circumstances, however, one may justify spending more time to make better predictions. Mass storage systems and theorem provers are examples of such applications, and in this paper we investigate the application of more sophisticated learning/prediction methods to the design of an MSS. The model is based upon the specifications for a proprietary MSS under development by a major high-technology company.

The MSS Model

Our model consists of a number of segments S, a cache with C slots each able to hold exactly one segment, and a number of users U. By assumption, $U < C < S$. Each segment has an associated read time, which is the *cost* of making a user wait while the segment is read into the cache. Each user is assigned a priority so that the waiting time of a high-priority user adds proportionally more to the cost.

A stream of user requests is generated by some hidden process. Each request is simply the name of the segment the user wishes to view paired with a length of time the user will spend before issuing his next request. This use time is relevant only to our simulation and is not available to the predictive algorithms described later. Since we are interested in situations where time is available to take advantage of sophisticated prediction, the use times are typically an order of magnitude larger than the read times, and both are measured in (simulated) seconds. Typical values are 10 seconds to read a segment into cache, and 200 or more seconds for the user to spend viewing the segment.

Whenever a user requests a segment not in cache, the user enters a waiting state where he remains until the read is completed. The user's waiting time multiplied by his priority factor is added to the system's total waiting time. The goal of any predictive algorithm is simply to minimize this total waiting time (cost).

The selection of the cache slot to replace on a read is of central importance. The segments currently in use are, of course, locked and not replaceable until a user requests a different segment. All other slots are kept in a queue whose head contains the next segment to be replaced. With a pure LRU strategy, the head is the segment least recently requested by any user. An alternative strategy for ordering this queue is explored in the following sections.

The TDAG Algorithm

A prediction algorithm can be used to develop online a probabilistic model of the request source, in order that the next few requests might be predicted and the contents of cache be adjusted accordingly. In this study we applied the TDAG sequence-prediction algorithm for this purpose. Based in part upon the Markov stochastic-process model, the TDAG algorithm can quickly learn complex sequential patterns in a stream of requests. It has been used successfully for other applications, including text compression and program optimization. See [7] and [8] for a detailed description.

Input to the algorithm is a stream $\{x_i \mid i \geq 1\}$ of symbols. Each symbol is treated as a distinct item and, in our application here, will be the name or address of a file segment. The algorithm builds a tree structure representing the most likely subsequences in the stream and, when data are sufficient to make a confident prediction, the algorithm predicts the next segment (or the next few segments) in the form of a partial probability distribution, e.g., "*Segment S_{72} with probability 0.53, or segment S_{23} with probability 0.38, or some other segment whose probability is too low to worry about*". The algorithm is designed to capture high-probability sequences without using valuable storage to record the myriads of low-probability events. Parameters control how much time and storage the algorithm can use in developing its model.

A Predictive Caching Algorithm

To implement a TDAG-based caching strategy, we create a separate TDAG for each user. On every segment request, the name of the segment is fed to the TDAG as its next input symbol. Over time, if there are stable patterns in the user's request sequences, the TDAG will be able to return a probability distribution on the next expected segment. This prediction can be used to compute an expected cost function $Cost(S_i)$ for each segment. The expected cost of not having S_i resident in the cache is given approximately by

$$Cost(S_i) = \sum_{u \in users} Prob_u(S_i) ReadTime(S_i) Priority(u),$$

where $Prob_u(S_i)$ is the probability that user u will request segment S_i, $ReadTime(S_i)$ is the time required to read S_i into cache, and $Priority(u)$ is the priority factor of user u.

This cost can now be used to rank the unlocked segments resident in cache. Segments whose probability of being requested is too small to predict are considered lowest cost and are scheduled for replacement by the default LRU mechanism. Furthermore, the predicted segment of highest expected cost not already in the cache can be prefetched if it will replace a segment of lower expected cost. We assume that prefetches can be cancelled immediately when a real user request comes in, so that the only cost associated with a prefetch is the danger of replacing the wrong segment.

For example, Figure 1 shows a cache with four slots. User U_1 is viewing segment S_{17} and U_2 is viewing S_{64}. U_2 has twice the priority of U_1. Projections and costs for some segments in the system are shown in the table. Because segment S_{85} has the highest expected cost and is not present in the cache, a prefetch would now be issued. It would be read into slot 3, replacing S_{42}, since of all (unlocked) segments in cache, S_{42} has the lowest expected cost.

Validation Experiment

To verify that our simulator works correctly and that we have implemented the model as intended, we ran a series of tests over a very simple scenario where it is possible to compute

Current user and cache state:

User	Priority	Current Segment	Slot
U_1	1	S_{17}	1
U_2	2	S_{64}	2
—	—	S_{42}	3
—	—	S_{51}	4

Segments (Sorted by Expected Cost):

Segment	Prediction: User	Probability	ReadTime	Expected Cost
S_{85}	U_1	0.64	15	9.60
S_{23}	U_2	0.38	10	7.60
...	...	...	...	...
S_{51}	U_1	0.07	11	0.77
S_{42}	(no prediction)		9	0.00

Figure 1: An example using predictions to determine which segment(s) to prefetch.

in closed form the expected waiting times. In this scenario there is one user and a cache with two slots—one in use and one available for prefetch. There are five segments S_1 through S_5, and to each is assigned a fixed probability P_i, $1 \leq i \leq 5$. Assuming the user is current viewing segment S_i, his next request is for segment S_j $(i \neq j)$ with probability $P_j/(1-P_i)$. This stochastic process can be represented by a Markov chain with five states; the TDAG algorithm can learn this process using a tree with 26 nodes.

Test cases were generated by assigning likelihoods and read-times to the five segments at random. It is fairly straightforward to compute the expected mean and standard deviation of the waiting time per request under the LRU (non-prefetch) strategy. The formula for the mean is given by

$$E(cost) = \sum_{i=1}^{5} Prob_{ss}(S_i) \sum_{j \neq i} Prob(S_j|S_i) \sum_{\substack{k, \\ k \neq i, k \neq j}} Prob(S_k|S_j) ReadTime(S_k).$$

Here, $Prob_{ss}(S_i)$ is the probability that, at a randomly selected instant, segment S_i has been viewed most recently. It is equal to the stationary probability distribution of the states in the Markov chain and is easily calculated using well-known techniques. $Prob(S_j|S_i)$ is the probability that a request for S_j follows that for S_i and, as noted above, is given by $P_j/(1-P_i)$. Given that S_i and S_j are the two segments in cache, we can compute the expected waiting time for the subsequent segment request; this is the third term in the formula and is a sum of the read times for each segment not in cache weighted by the probability of its being requested. Note that no waiting cost is incurred if segment S_i (which is already in cache) happens to be the next to be requested.

Now consider the situation when segments can be prefetched. It is easy to show that the optimal prefetch strategy is to always read the segment S_i with the highest product $P_i \times ReadTime_i$ unless it is already being viewed, in which case we should fetch the segment with the next highest product, $P_j \times ReadTime_j$. With this observation we can also compute the expected mean and standard deviation of the waiting time under an optimal prefetch strategy.

For any given probabilities and waiting times, we can compute and compare the waiting times for both the LRU and TDAG-prefetch strategies. This provides an exact comparison of the two strategies under very special circumstances, but the main purpose of studying this simple case is to verify that the experimentally observed waiting-time statistics agree with the predictions and thereby to build our confidence in the accuracy of our simulation. We can then proceed to simulate more complex (and more realistic) user request processes and use our model as a low-cost way to compare the TDAG-prefetch strategy with other possible strategies.

Segment Statistics:

Segment (S_i):	S_1	S_2	S_3	S_4	S_5
Prob. (P_i):	0.141	0.272	0.123	0.170	0.294
ReadTime (T_i):	10.13	18.42	12.89	11.37	14.12
$P_i \cdot T_i$:	1.43	5.01	1.59	1.93	4.15
Stationary Prob.:	0.156	0.255	0.139	0.182	0.268

Theoretical vs. Experimental Results:

	Mean	Std. Dev.	Number of Misses*
Expected cost (LRU):	9.858	6.622	–
Expected cost (TDAG):	7.765	5.928	–
Observed cost (LRU):	9.914	6.593	7252
Observed cost (TDAG):	7.918	6.062	6455

*A "miss" is a request for a segment not in any of the five slots.

Figure 2: Typical experimental results for the simple single-user, five-slot model.

We generated over 650 test cases at random, computed the expected statistics, and then ran our simulator for 10000 requests in LRU strategy and 10000 requests using TDAG prefetching. Figure 2 shows the inputs and results for a typical test case. Also shown is the number of times during the simulations that a requested file was not available in cache. During the run with prefetch, this case issued 2138 prefetch requests, of which 1133 successfully predicted the user's next request.

This run matched others in providing quite good agreement between theory and experiment. With the TDAG strategy we consistently observed an average of about 3 percent

	Case 1	Case 2	Case 3
Number of Users	10	6	2
Segments per User	3	5	15
Total Requests per Run	30000	30000	30000
Mean Waiting Time (LRU)	15.21	10.29	5.63
Mean Waiting Time (TDAG)	11.74	3.80	0.44
Waiting Time Reduction	23%	63%	92%
Demand Faults (LRU)	29948 (.998)	23291 (.78)	12886 (.43)
Demand Faults (TDAG)	25189 (.84)	9945 (.33)	1344 (.05)
Prefetch Fault Reduction	16%	57%	90%
Prefetches Issued	7408	18923	71714
Prefetch Transfer Ratio	1.09	.96	2.44

Figure 3: Experimental results with multiple users and 12-slot cache

more waiting time than the theory predicts. Although the exact nature of the discrepancy has not been carefully studied, exact agreement is not expected since the TDAG is subject to statistical fluctuations in its learning and prediction results.

Averaged over all runs, the TDAG-prefetch strategy yielded a cost savings of about 15 percent over LRU. In some extreme cases, the savings were as high as 48 percent, and in others there were no savings at all. The greatest savings occured in the cases where one or two segments stood out as having highest cost. No savings were possible when all the segments had essentially equal expected costs.

Other Experiments

For our main experiment we used a cache with 12 slots and a pool of 30 segments. The number of users covering these 30 segments was varied: first 10 users were assigned 3 segments each; then six users were given five segments; and finally 2 users were each assigned 15 segments. The users' request sequences were generated by random stochastic finite automata. Each automaton had either 2 or 3 states per output symbol (segment), was fully connected and had an out-degree of 3. Thus the request sequences formed a complex pattern, but of the sort in which we expect TDAG to be able to find regularities. For simplicity, all users were assigned equal priorities so the cache manager's task is purely to minimize waiting time.

Each test consisted of two lengthy runs using the same set of users. Large numbers of requests were used to ensure statistical significance of the results. The first run was managed only by the default LRU strategy. However, during this run, a TDAG silently observed each user and built its representation of that user's request pattern. On the second run, these already trained TDAGs were allowed to make predictions for the prefetch mechanism. The

cache manager then issued prefetches to fill the unlocked slots with the segments of highest expected cost. Thus the differences in the statistics between the runs is a best-case measure of how well TDAG prefetching can perform given these users with this configuration.

Sample results are shown in figure 3. We first call attention to the Mean Waiting Time figures because, ultimately, this is the statistic that our prefetching strategy was aimed at reducing. As before, these numbers are in (simulated) seconds per user request, and the percentage improvement with prefetching is noted.

Previous literature on prefetching has often emphasized the *miss ratio* and the *transfer ratio* as performance measures. The *miss ratio* is simply the number of demand faults divided by the number of requests, i.e. the fraction of the time a user had to wait for a read. The miss ratio is noted in parentheses next to the fault counts. The *transfer ratio* is the total number of reads, both demand fault and prefetch, divided by total requests. Clearly, without prefetching the miss and transfer ratios are equal, while with prefetching the transfer ratio will rise somewhat while the miss ratio, hopefully, goes down.

In our simulation model, no cost was incurred for issuing a prefetch. This is reasonable, assuming, as we were, that the MSS would have dedicated, interruptible hardware to handle the reads. Therefore, we are relatively unconcerned with how much the transfer ratio rises, but perhaps the dramatic rise to 2.44 in test case 3 deserves some discussion. In this case, there were ten free slots which the cache manager could reconfigure after each request by one of the two users. Given these parameters, one might expect a transfer ratio as high as about 5.0. Other applications may have different cost considerations, and these should be taken into account when designing the prefetch decision strategy.

To test the learning characteristics of TDAG we used the same user models and system parameters. This time, however, the learning and prediction were intermixed so that prefetching began as soon as the TDAGs had built enough of a model to begin issuing predictions. Figure 4 shows the miss ratio improvement under these conditions. These data were collected by recording the run statistics after every 100 user requests, so the curves are quite noisy. However, the overall trends are clear. After the first 3000 requests or so, each curve has found its performance level and remains in the same range thereafter. These are total request figures so 3000 represents approximately 300 training instances per TDAG in the 10-user case and 1500 training instances in the 2-user case. Greater learning times per user were to be expected in the 2-user case due to the greater complexity of their Markov chains (30 states versus 9).

We ran other experiments with varied parameters that served to confirm the obvious intuitions about our cache manager scenario: Having a greater proportion of free slots is always advantageous to both the LRU and Prefetch strategies; and, given a deterministic user model, say a simple cycle among a set of segments, TDAG will become a virtually perfect predictor and improve performance up to the limit imposed by the cache size.

However, it is worth noting that one can easily build a non-Markovian user model advantageous to LRU as follows: After the first two requests, with probability p (say .8) reselect the most recently used previous segment, and with probability $1-p$ select uniformly at random from the entire set of segments. Clearly, LRU is optimal in this case, and TDAG can

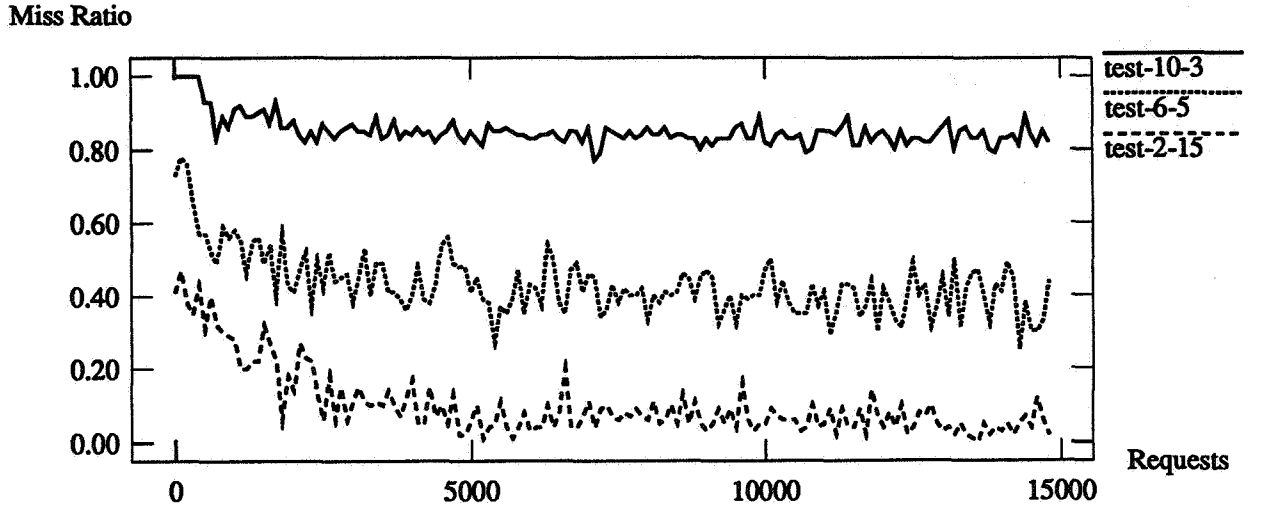

Figure 4: Miss Ratio Change with Learning.

only learn spurious relationships with low probability of being repeated. Using this model, we were able to trick the prefetch strategy into actually *degrading* performance by up to 95 percent. Therefore, while TDAG learns well for a large variety of sequence types, designers of database systems should study empirically the characteristics of their expected request sequences before committing to a prediction strategy.

Discussion and Related Work

We have not yet made mention of the computation overhead that our prefetching strategy entails. This was not a major consideration in this study, in keeping with our assumption of a relatively slow, dedicated MSS. However any real system will have some resource limits it must abide by. Our simulation environment was not amenable to collecting meaningful computation measures. However, the TDAG algorithm is designed so that one may adjust the tradeoff between speed and accuracy according to the available computational resources. Parameters control the amount of storage available for remembering contexts and the maximum turnaround time (the time between arrival of an input symbol and the output of the prediction for the next symbol). Parameters also control the prediction risk, i.e. how much evidence is required before a prediction can be made on the basis of a given context.

Those familiar with the details of TDAG as described in [8] may be curious about the parameter settings used in this study. For the record, the data presented here were collected using the "Lazy" strategy, an extendibility threshold of 20, a projection threshold of 50 and

a maximum height of 11. The same tests were run multiple times with different random number seeds and repeated using the "Eager" strategy, which is expected to learn somewhat faster at the expense of more memory usage. The final results did not vary significantly and the numbers shown here are representative.

There have been few attempts to implement significant prefetching strategies in virtual memory systems [10]. However, the idea of prefetching has been around for a long time. Many models are based on the idea of sequentiality, especially in the work of Smith [15, 14, 16]. Sequentiality is the observation that, due to the program or database structure, a reference to page i is often a strong predictor of page $i + 1$. Other research has explored the use of user-provided or compiler-generated predictions of page references [17, 18, 4, 1]. These studies have all supported the conclusion that good time savings can be gained by prefetching, provided that a reliable predictor of future references exists. Our work aims at learning these predictions automatically and we note that TDAG can readily learn sequential reference patterns if it is given relative segment *offsets* rather than absolute addresses.

A somewhat later study, and closer in spirit to the present approach, is the work of Lau [9] and Martinez [11]. These authors propose very similar empirical predictors which are trained from traces of page requests. A matrix $E(i, j)$ records the number of faults to page j that occured within some parameter T references after a fault to page i. This matrix is then used to rank the most probable future faults given a current fault. Lau combines this prediction with the sequential model and considers fetching one extra page. Martinez weights his predictions by the expected time until the next fault and considers models which prefetch the most likely p pages. Both show encouraging results from their trace-based simulations. While our cost model required learning the total request sequence regardless of faults, TDAG could also serve as a tool for this model by feeding it input only after faults.

Horspool and Huberman [3] point out the desirability of maintaining the *memory inclusion property*. This property simply states that the pages a policy retains in memory is always a subset of the pages that policy would retain given a larger memory. The lack of this property leads to the famous FIFO anomaly and makes it difficult to predict how a policy will scale to different memory sizes. Furthermore, Horspool and Huberman observe that no previous prefetching scheme with which they were familiar possesses this property. The method described in this paper uses a priority allocation scheme which ensures the memory inclusion property.

More recent work has also demonstrated the desirability of anticipatory fetching (e.g. [5]). Several researchers are exploring models similar to ours, requiring sequence prediction modules. Palmer and Zdonik [12] use a nearest-neighbor pattern matcher to predict possible sequence continuations. Salem [13] builds on the sequential approach by building a table to record the times segment y followed segment x, cleverly using parameters to limit storage requirements and remember the most useful relations.

Finally, Vitter and Krishnan [19] have applied the Ziv-Lempel compression algorithm to the prediction task and have proved that for an ergodic Markov source the resulting online prefetching algorithm is asymptotically optimal. Recent work by the same authors [6] has strengthened this result. A practical application of these ideas is described in [2]. The authors

built a simulator very similar in essence to ours and ran it on benchmark database access traces with arguably more realistic memory size parameters. They show excellent miss ratio improvements, in the 20 to 60 percent range, especially with a Markov tree text-compression algorithm called PPM that is closely related to TDAG.

Conclusions

Predictive caching, which uses predictions about which storage elements will be requested in the future in order to decide which of the currently resident segments to replace, is simple and effective using the TDAG sequence-prediction algorithm. The simplicity of the TDAG model also enables us to compute theoretically the expected performance of the caching system under some circumstances. Our own calculations agreed quite well with simulation results based upon the model of a mass storage system. Further simulations give evidence for the large cost savings that can theoretically be gained using adaptive prefetching.

References

[1] Glen Alan Brent. *Using Program Structure to Achieve Prefetching for Cache Memories.* PhD thesis, University of Illinois at Urbana-Champaign, 1987.

[2] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. Manuscript received Dec. 1992.

[3] R. Nigel Horspool and Ronald M. Huberman. Analysis and development of demand prepaging policies. *The Journal of Systems and Software*, 7(3):183–194, 1987.

[4] Eric E. Johnson. Working set prefetching for cache memories. *Computer Architecture News*, 17(6):137–141, 1989.

[5] D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Distributed and Parallel Databases*, 1:33–51, 1992.

[6] P. Krishnan and J. Vitter. Optimal prefetching in the worst case. Manuscript received Dec. 1992.

[7] Philip Laird. Discrete sequence prediction and its applications. In *Proc., 9th National Conference on Artificial Intelligence.* AAAI, 1992.

[8] Philip Laird. TDAG: An algorithm for learning to predict discrete sequences. Technical Report FIA-92-01, NASA Ames Research Center, AI Research Branch, 1992.

[9] Edwin J. Lau. Improving page prefetching with prior knowledge. *Performance Evaluation*, 2(3):195–206, 1982.

[10] Mamoru Maekawa, Arthur Oldehoeft, and Rodney Oldehoeft. *Operating Systems: Advanced Concepts.* Benjamin/Cummings, 1987.

[11] Michel Martinez. Program behavior prediction and prepaging. *Acta Informatica*, 17:101–120, 1982.

[12] M. Palmer and S. B. Zdonik. Fido: a cache that learns to fetch. In *Proceedings of 17th International Conference on Very Large Data Bases*, 1991.

[13] Kenneth Salem. Adaptive prefetching for disk buffers. Technical Report Tr-91-46, University of Maryland and CESDIS, Goddard Space Flight Center, 1991.

[14] Alan J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–20, 1978.

[15] Alan J. Smith. Sequentiality and prefetching in database systems. *Transactions on Database Systems*, 3(3):223–247, 1978.

[16] Alan J. Smith. Cache memories. *Computing Surveys*, 14(3):7–20, 1982.

[17] Kishor S. Trivedi. Prepaging and applications to array algorithms. *IEEE Transactions on Computers*, C-25(9):915–921, 1976.

[18] Kishor S. Trivedi. An analysis of prepaging. *Computing*, 22:191–210, 1979.

[19] J. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 1991.