Marginal Cost-Benefit Analysis for Predictive File Prefetching

Timothy Highley
Department of Computer Science
University of Virginia
tjhighley@cs.virginia.edu

Paul Reynolds
Department of Computer Science
University of Virginia
reynolds@cs.virginia.edu

ABSTRACT
File prefetching can reduce file access latencies and improve overall performance. Prefetching can be especially important in on-line software on demand, where network latencies create unacceptable delays. Prefetching involves predicting future accesses and establishing when/whether to prefetch, based on future access predictions. Cost-benefit analysis (CBA) [15] addresses when/whether to prefetch and it addresses the interaction between prefetching and caching. CBA weighs the expected benefits of file prefetching and the cost of expected buffer usage. We describe 1-Marginal CBA, an approach that employs probabilistic predictions, as opposed to deterministic hints [16]. We present a probabilistically optimal, though intractable, algorithm, Opt, for a representative prediction model, and demonstrate that any other optimal algorithm under that model will also be intractable. We argue that in many circumstances 1-Marginal and Opt will make the same decisions. Finally, we present simulation results in which 1-Marginal reduced I/O time by an average of 19% and a maximum of 49% over other prefetching schemes in the literature, even when using the same predictor. Since the cost of 1-Marginal is comparable to that of other published algorithms the improvement is real.

Categories and Subject Descriptors
D.4.8 [**Operating Systems**]: Performance – *modeling and prediction, simulation.* F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *computations on discrete structures.*

General Terms
Algorithms, Measurement, Performance, Design, Experimentation.

Keywords
Prefetching, cost-benefit analysis, performance.

1. INTRODUCTION
Processing speeds have improved much more quickly than disk access speeds and network latencies. As a result, improvements in file system access times have become increasingly important.

File prefetching has been widely researched in order to address the problem of slow disks, and particularly disk access latency. Much of the research has focused on how to best predict future accesses [2][5][13][10][12], though some researchers [4][15] have examined the question of when to perform prefetches, assuming knowledge of future accesses. Patterson introduced cost-benefit analysis for file prefetching, which seeks a good balance between caching recently used blocks and prefetching blocks which are predicted for future use (Figure 1). The approach was extended to systems that assume only a probabilistic knowledge of future accesses in [18]. In [7], distinctions were drawn between two types of cost-benefit analysis: absolute and marginal. An analysis of absolute CBA was also presented there.

Our research follows [15], [18] and [7]; we present an algorithm for file prefetching using marginal cost-benefit analysis. This approach employs the concept of marginal gains, similar to the approach used in [14], [17] and [8]. Marginal gains are the benefits obtained by taking actions earlier than they otherwise would have been taken, or allocating more resources than otherwise would have been allocated. Previous researchers partitioned the file cache and used marginal gains to establish how many buffers to assign to each partition. In [17] each partition corresponded to a different process, while in [8] each partition corresponded to a different access pattern. Our approach uses the concept of marginal gains on a per-block basis so that the most time-critical blocks are prefetched.

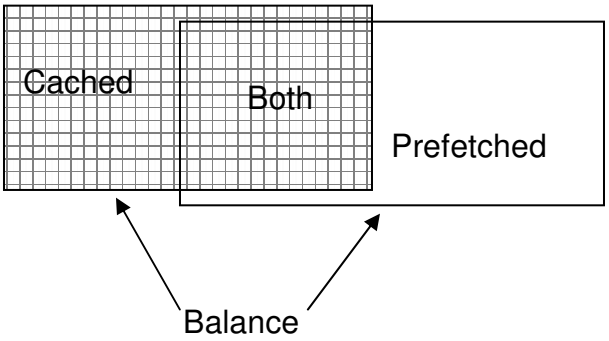

Figure 1. CBA: seeking a balance between caching and prefetching for file buffers.

We also present Opt, an algorithm that, when provided with a finite probability tree representing all possible executions of the program, will prefetch optimally in the average case. A runtime lower bound for any optimal algorithm on a similar prediction model is also presented.

Our results capture an effective approach for practical prefetchers. Our simulations indicate that 1-Marginal CBA can reduce I/O time by 19% compared to other prefetching algorithms.

2. ABSOLUTE VS. MARGINAL COST-BENEFIT ANALYSIS

Cost-benefit analysis identifies which file blocks to place and hold in a file buffer. Estimators express the cost or benefit of ejecting or fetching particular blocks. Estimators are expressed in terms of a common currency, which takes two factors (program execution time and cache buffer usage) and expresses them as one number that can be compared for all potential blocks. If the benefit of initiating a fetch ever exceeds the cost of ejecting a block currently in the cache, the fetch is initiated [15].

In [7], a distinction is drawn between absolute and marginal CBA. In absolute CBA, the benefit of a particular action is related to the difference between taking the action and not taking the action. In marginal CBA, the benefit of a particular action is related to the difference between taking the action immediately and the effect of waiting to take the action at a later point in time. If that later point in time is assumed to be the next opportunity, we call it 1-Marginal CBA.

Absolute CBA is somewhat simpler than marginal CBA, and in systems with very little prefetching it may perform well. With absolute CBA the question asked is: "If only one block is ever fetched, which one should it be?" However, in most cases several blocks are good prefetch options, and in the case of probabilistic prefetching many predictions will be incorrect. Frequent mispredictions imply that, often, buffers become available for other uses. In this scenario the 1-Marginal assumption is likely to be true: a prefetch that is not initiated immediately can be initiated after the next file block access. In 1-Marginal CBA the question asked is: "Which block should be fetched if other blocks can be prefetched in the next access period?"

3. SYSTEM MODEL

For our analysis, we assume a program consisting of $\mathbf{N_{I/O}}$ file accesses, with a constant amount of processing, $\mathbf{T_{CPU}}$, after each access. The time to satisfy a file access when the file resides in the file block cache is $\mathbf{T_{hit}}$. If the file block must be fetched from the disk, it requires $\mathbf{T_{driver}}$ CPU time to prepare the fetch, and $\mathbf{T_{disk}}$ for the block to arrive in the file block cache. $\mathbf{T_{stall}}$ is the time the processor spends waiting for the file block; it is equal to $(T_{disk} + T_{hit})$ if the block is not prefetched or cached, but it may be less if it has been prefetched recently or is currently cached. An *access period* consists of a file access request, a series of file fetching decisions, the access of the requested file, and processing until the next file access request.

We assume that a prediction mechanism provides an accurate probability tree, rooted at the most recently requested block, that corresponds to potential future accesses. By accurate, we mean that the probability associated with a block in the tree corresponds to the unconditional probability of accessing the block. In a real system, the predictor will not be perfect; accurate predictors are an open problem. The tree may be incomplete (i.e. the outbound edges on a given node may sum to less than one). Predictors appearing in the literature (e.g. [5][6][13]) could be used to provide an adequate probability tree.

We use established terminology when discussing nodes and their relationships in a probability tree: root, leaf, predecessor, immediate predecessor, descendant and immediate descendant.

4. PREFETCH ESTIMATOR FOR 1-MARGINAL CBA

We describe an estimator, **E**, for evaluating the benefit of prefetching a block in 1-Marginal cost-benefit analysis. **E** is the expected change in total I/O time divided by the expected change in buffer usage (or bufferage [15]).

For a block **b** that may be potentially prefetched, $\mathbf{d_b}$ denotes its depth (or distance) in the probability tree, where depth is measured as the number of accesses that would precede **b**, if the path to the block were to be followed. $\mathbf{p_b}$ is the probability that the block will be accessed. **s** is the average number of prefetches, per access period, in which T_{driver} for the prefetch is not overlapped with disk stall.

A prefetch is *correct* if the block fetched is accessed when it was predicted to be accessed. If a prefetch is correct, then any processor activity before the block is needed (up to T_{disk}) will mask the stall for the block. In addition, if the processor is currently stalled, then T_{driver} for the prefetch is overlapped with that stall time. Let $M(r, b, d_b)$ denote

$$\begin{aligned}&\max(-T_{driver}, -T_{stall}(r)) + \\ &\max\left(-d_b * (T_{CPU} + T_{hit} + sT_{driver}) - \right. \\ &\qquad [(\textstyle\sum_{m = b\text{'s direct ancestors starting with } r} T_{stall}(m)) + \\ &\qquad \left.\max(-T_{driver}, -T_{stall}(r))],\ -T_{disk}\right) \qquad (1)\end{aligned}$$

$M(r, b, d_b)$ represents the amount of time a correct prefetch of b will save if b is prefetched at a distance of d_b with r as the root at the time of the prefetch. The amount of time is negative to indicate a reduction in I/O time.

The first term of equation *(1)* indicates that if the processor is currently stalled, then a prefetch can mask its T_{driver} cost with the current stall time. The second term indicates that stall time for b can be overlapped with stall time and processing time on accesses that come before b. It also indicates that T_{driver} for future prefetches can be overlapped with the stall time for the current prefetch. In order to avoid double-counting, s does not take into account all prefetches, but only those where T_{driver} is not overlapped with disk stall.

If a prefetch is incorrect, then the time to prepare the fetch, T_{driver}, was wasted, but if the processor was stalled anyway there is no effect on the total I/O time. If a block is prefetched then the expected impact on I/O time is

$$p_b * [M(\text{root}, b, d_b)] + (1 - p_b) * (\max(T_{driver} - T_{stall}(\text{root}), 0) \qquad (2)$$

In *(2)*, 'root' is the root of the probability tree (i.e. the most recently requested block).

By waiting one access period before prefetching, one of three scenarios occurs. 1) If the next block requested does not lead toward the potential block, b, then b will not be prefetched at all and a misprediction will be avoided (and thus the program runtime is unaffected). 2) If b is eventually accessed, then the prefetch at the next access period will reduce the stall time, but by a smaller amount than if the block had been prefetched immediately. 3) b may still be mispredicted, adding T_{driver} to the program runtime. Define p_{b1} to be the probability that one access period later b is still a candidate for being prefetched. That is, b is a descendant of the new root node one access period later. The impact of a prefetch that might be performed one access period later is

$$(1 - p_{b1}) * (0) + p_b * [M(\text{next node}, b, d_b - 1)] + (p_{b1} - p_b) * (\max(T_{driver} - T_{stall}(\text{next node}), 0) \quad (3)$$

In *(3)* 'next node' refers to the node that is an immediate descendant of root and is either a predecessor of b or is b itself. The difference between *(2)* and *(3)* is the amount of time saved that can be attributed to prefetching one access period earlier:

$$p_b * [M(\text{root}, b, d_b)] + (1-p_b) * (\max(T_{driver} - T_{stall}(\text{root}), 0) - [(1 - p_{b1}) * (0) + p_b * [M(\text{next node}, b, d_b - 1)] + (p_{b1} - p_b) * (\max(T_{driver} - T_{stall}(\text{next node}), 0)] =$$

$$p_b * [M(\text{root}, b, d_b) - M(\text{next node}, b, d_b - 1)] + (1-p_b) * (\max(T_{driver} - T_{stall}(\text{root}), 0)) - (p_{b1} - p_b) * (\max(T_{driver} - T_{stall}(\text{next node}), 0) \quad (4)$$

Buffer usage is measured in units of "buffer-access" [15], which is the occupation of a file buffer for one access period. In deriving E the issue is prefetching a block one access period earlier, so the bufferage is one buffer-access and the benefit of prefetching block b at depth d_b is

$$B(b) = [p_b * [M(\text{root}, b, d_b) - M(\text{next node}, b, d_b - 1)] + (1-p_b)(\max(T_{driver} - T_{stall}(\text{root}), 0)) - (p_{b1} - p_b)(\max(T_{driver} - T_{stall}(\text{next node}), 0)] / 1 \quad (5)$$

This is the 1-Marginal benefit estimator, E, for prefetching block b.

Formula *(5)* can be simplified without having a significant impact on performance. If 'next node' is assumed to be cached then T_{stall}(next node) is zero. If it is also assumed that the enforced maximum of $-T_{disk}$ in $M(r,b,d_b)$ is unnecessary, then part of the formula simplifies so that d_b is no longer a factor. This is a reasonable assumption in many cases; the maximum only becomes relevant when a prefetch is performed at the prefetch horizon. The simplification is:

$$(M(\text{root}, b, d_b) - M(\text{next node}, b, d_b - 1)) =$$

$$\max(-T_{driver}, -T_{stall}(\text{root})) + -d_b * (T_{CPU} + T_{hit} + sT_{driver}) - [(\sum_{m = b\text{'s direct ancestors starting with root}} T_{stall}(m)) + \max(-T_{driver}, -T_{stall}(\text{root}))] - [-(d_b\text{-}1) * (T_{CPU} + T_{hit} + sT_{driver}) - [(\sum_{m = b\text{'s direct ancestors starting with next node}} T_{stall}(m))]] =$$

$$-(T_{CPU} + T_{hit} + sT_{driver}) - T_{stall}(\text{root}) \quad (6)$$

Also, when a block is incorrectly prefetched, the T_{driver} overhead may prevent a successful prefetch from being initiated, even if T_{driver} is overlapped with stall time. This gives some justification to the decision to assume that T_{driver} overlapped with stall time should still count as a penalty. By making that assumption, the last two terms in the numerator of *(5)* can be combined. With these simplifications, the benefit can be expressed as

$$B(b) = p_b(-(T_{CPU} + T_{hit} + sT_{driver}) - T_{stall}(\text{root node})) + (1-p_{b1})(T_{driver}) \quad (7)$$

This is the simplified 1-Marginal (S1M) benefit estimator for prefetching block b.

Because the root node is the same for all blocks that might be prefetched, p_b and p_{b1} are the only discriminating factors in deciding which block to prefetch. In cases where T_{driver} is negligible, p_b is the only factor in determining the most beneficial block. The S1M benefit estimator is important, though, in deciding when to prefetch, since the prefetch estimator is compared with other estimators such as the one for retaining blocks in the demand cache.

The above derivations for 1-Marginal and S1M are based on the assumption that each node in the tree represents a different file block. It is possible that a single block may be replicated several times in the probability tree, and that the block may be the best prefetch choice because it appears several times in the tree, even though the individual probabilities associated with the nodes are small.

With replication, the analysis changes only slightly. For block b, p_b becomes the sum of the probabilities of all instances of nodes representing b (provided the node is not a descendant of another instance of a node representing b). The depth of b is no longer clearly defined, since the nodes may be at different depths. Fortunately, in the simplified formula, d_b drops out of the benefit equation because it is only important to note that by waiting to prefetch, the depth of each node is reduced by one, just as the depth is reduced by one in the non-replicated case. The definition for p_{b1} remains the same: the probability that one access period later b is still a candidate for being prefetched. Thus, the S1M CBA benefit estimator for prefetching in the replicated case is expressed the same as that in the non-replicated case, but p_b and p_{b1} are calculated differently in the replicated case.

In [7], absolute CBA with replication was investigated. An anomaly described there also applies here. Though it may be counterintuitive, it is possible that considering additional nodes for a particular block artificially reduces the estimated benefit of prefetching the block. To counteract this, the benefit of prefetching a particular block is the maximum value of B(J) where J is a subset of the nodes representing block b.

In a previous study, we found that absolute CBA realized no significant improvement with the addition of replication while the cost of computation was prohibitively high. It is possible that replication would have greater benefit in 1-Marginal or S1M CBA, but experience with absolute CBA indicates otherwise. As disk speeds and processor speeds continue to diverge, replication may eventually become a more important issue.

1-Marginal analysis could also be applied to derive estimators about ejecting blocks from the cache. These estimators would not be practical, however, because it would not make sense to eject a block and fetch it again at the next access period.

5. OPTIMAL PREFETCHING ON A FINITE TREE

In [1], an algorithm is presented to calculate the optimal prefetching/caching schedule for a single disk problem where the entire request sequence is given in advance. In [19], a prefetching algorithm is presented that is optimal in the limit for *pure prefetching*. Pure prefetching is when there is time between each access to prefetch as many blocks as desired. In [9], an algorithm is presented which optimizes for the worst case, again under the assumption of pure prefetching. With pure prefetching, the prefetching problem reduces to a prediction problem. The literature does not currently offer an optimal algorithm for non-pure prefetching in the presence of uncertainty. For the new algorithm we present here, Opt, we assume that the prediction problem is solved. That is, a finite probability tree is provided that describes all possible executions of the program. Opt is optimal in the average case, assuming zero-cost analysis. (Note that for this problem, the LRU cache is not taken into consideration since all possible accesses are represented in the tree.) Opt is intractable in both run-time and space used, but it is worthwhile to ask: "What should be prefetched here?" and have a methodology for determining the correct answer.

We first introduce the concept of an *opt-mapping*, which is a mapping, for a particular tree node, from cache state to expected execution time. The cache state is defined by the collection of blocks in the cache, the blocks that are currently being fetched into the cache and the arrival times of those blocks being fetched. The opt-mapping for a particular node of a probability tree is a mapping from cache state to expected time to complete execution on the subtree of which that node is the root, when probabilistically optimal prefetching decisions are made on that subtree. If an opt-mapping is found for each node of the tree, then probabilistically optimal prefetching can be performed on the tree. We use $om(n, cs)$ to denote the value of the opt-mapping at node n when the cache state is cs. For example, om(root node, empty cache) is the expected execution time for the entire program when optimal prefetching is employed.

If we have an opt-mapping for each child of a parent node, an opt-mapping can be constructed for the parent node. To determine the value of $om(P, cs)$ for a particular parent node P and cache state cs, first determine the time needed to access and perform processing on P's block. If, for cs, the block is in the cache, the time needed is (T_{hit} + T_{CPU}). If the block is not in the cache at all, it is (T_{driver} + T_{disk} + T_{hit} + T_{CPU}). If the block has been fetched and is on its way to the cache, it is ((Time until block arrives) + T_{hit} + T_{CPU}). Define $acc(b(n), cs)$ as the time needed to access and perform computation on the block represented by node n when the cache state is cs. Second, determine the set $T(cs)$ of all possible cache states to which it is possible to transition from cs. Also compute the times needed for the transitions (i.e. T_{driver} for each fetch initiated with possible overlap for stall time and computation time). Define $\tau(cs, t)$ as the time needed to change the cache from state cs to state t. For each possible transition state t, add the access time, time needed for the transition and the value of the immediate descendants' opt-mappings, each evaluated at t and weighted for probability of following that edge of the probability tree. Find the transition state m for which the sum of all these (the access and computation time, the penalty and the weighted opt-mappings, evaluated for each immediate descendant) is the least. That sum is the value of $om(P, cs)$. The prefetching decisions that should be made at the parent, if the initial cache state is cs, are those decisions that will transform the cache state to m. More formally,

$$om(P, cs) = \quad (8)$$
$$acc(b(P)) + \min_{t \in T(cs)}[\tau(cs, t) + \sum_{i=P\text{'s children}}(p_i * om(i, t)]$$

Given this recursive method for constructing an opt-mapping, we can construct an opt-mapping for the root node if we can construct an opt-mapping for the leaf nodes. An opt-mapping for a leaf node is trivial. A single node represents a program with one disk access. For any cache state that does not include the node's block in the cache at all, the expected execution time is (T_{driver} + T_{disk} + T_{hit} + T_{CPU}). For any cache state that has the block in the cache, the expected execution time is (T_{hit} + T_{CPU}). For any cache state that has a buffer reserved for the block, with the fetch initiated but pending, the expected execution time is ((Time until block's arrival) + T_{hit} + T_{CPU}). The only fetching decision to be made here is whether or not to fetch the leaf node's block if it is not already in the cache. It should always be fetched; this is the optimal action to take regardless of the previous cache state.

Given an opt-mapping for the root node, the initial prefetching decisions are those suggested by the opt-mapping evaluated at the empty cache state.

The concept of a prefetch horizon is introduced in [15]. The prefetch horizon is the depth of the access that is furthest in the future for which there may be some benefit to prefetching. Accesses beyond the prefetch horizon may be prefetched one access-period later and still arrive in the buffer in time to incur no stall.

We conjecture that 1-Marginal is optimal whenever a sequence of file accesses does not lead to prefetches that are as deep as the prefetch horizon. Proof is left to future work, as is a characterization of the effects prefetches at the prefetch horizon may have on 1-Marginal's performance relative to that of Opt.

6. BOUND ON EFFICIENCY OF AN OPTIMAL PREFETCHER

Opt is not practical for a real system, and it is natural to inquire whether a prefetcher could make an optimal prefetching decision without evaluating the entire tree. The following counter-example demonstrates that any prefetcher that is probabilistically optimal must in some cases examine the deepest leaf node.

For our counter-example, we assume a cache size of 4 and a prefetch horizon of 2. In both Figure 2 and Figure 3, blocks a, b and d should be fetched immediately. In Figure 2 the fourth block fetched should be block c. In Figure 3 it is more beneficial to prefetch block e instead of block c. The difference is the presence or absence of block n. However, the presence or absence of block

n is not detectable unless the algorithm considers that leaf node at the deepest depth of the tree. This example deals only with a maximum depth of four, but is extensible to any depth, where the probability of accessing block m is 99.9 per cent (or 99 per cent in the case of Figure 3), the probability of taking the initial branch to the right is 10^{-n}, where n is the maximum depth minus one, and the probability of taking any of the other branches is 9 x 10^{-n}, where n is the maximum depth plus one, minus the depth of the node being branched to. This demonstrates that the problem of determining an optimal prefetching decision with a finite tree prediction model is Ω(n), where n is the depth of the tree.

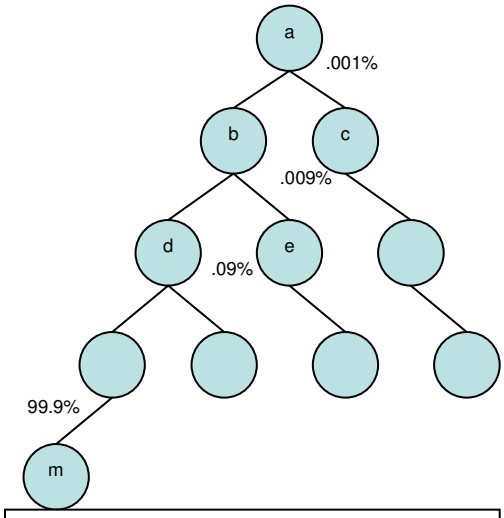

Figure 2.

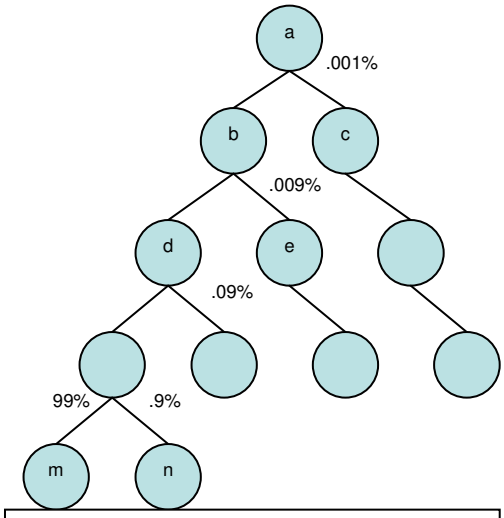

Figure 3.

7. PERFORMANCE RESULTS

There are many possible prediction models. A perfectly accurate finite probability tree lends itself well to analysis but is impractical for production level implementations. In our simulations we chose to use a context model predictor [5], which can emulate a probability tree. A full rationale for our prediction model selection can be found in the appendix.

7.1 System Model and Traces

The model of our simulator faithfully represents the theoretical model discussed in this paper, with the exception of the predictor. Because we used a context-model instead of a finite tree, the probability tree is potentially infinite. Also, the theoretical model is based on a perfect predictor, but the simulator's predictor is imperfect. We chose system parameters representative of a typical desktop workstation. Our model assumes the same type of processor as [15], so we use similar values, but scaled for current desktop workstations: T_{hit} = 5.8 microseconds and T_{driver} = 10.5 microseconds. The values for T_{disk} and T_{CPU} were not the same for all of our simulations, but unless otherwise stated their values were 10 milliseconds and 100 microseconds respectively.

Our prediction mechanism is an implementation of a second order Markov predictor [5], using escape probability method A with lazy exclusions, found in [3], p. 144. That is, we consider only the two most recent accesses, and predict future accesses based on what has previously happened immediately following those two accesses. If a particular block has not been previously seen in the context of the two most recent accesses, then the escape probability is used to determine a probability for that block, based on lower-order contexts. For efficiency, in order to prevent probing along a large number of paths at the same time, the prefetching system considers at most twelve descendants at depth one. At subsequent depths, at most one immediate descendant per node is considered for prefetching. This means that at any point in time, there could be prefetching along twelve different branches of the probability tree, to a depth limited only by the prefetch horizon. We use the same predictor with each of three prefetch estimators in order to see the impact of the prefetch estimators. We are not attempting to evaluate the performance of the predictor.

Our traces are the cello, snake and CAD traces from [18]. The cello and snake traces were previously used in [16] and the CAD trace was previously used in [5]. Cello is a trace of a timesharing system and snake is a trace of a file server. Both are traces of accesses to the file system, so requests that hit in the cache are not part of the trace. CAD is a trace of object references (rather than file block accesses) from a CAD tool. More details about the traces are available in the papers cited.

7.2 Results

For each trace, we report results using three different prefetching estimators. Two of the estimators are those described in this paper: 1-Marginal and Simplified 1-Marginal. The third estimator is the prefetch estimator from [18], which we will call VC. The performance of VC was shown to be comparable to the best performances of two approaches that rely on tunable parameters [11][5], but without relying on tunable parameters itself. This is important because the optimal tuning for a parameter can change from one set of data to another.

We vary T_{disk} from 5 ms to 50 ms, representing latencies from modern disk drives to slow DSL lines. We vary inter-access processing time from 100 microseconds to 20 milliseconds. We vary cache size from 10 buffers to 2048 buffers. We also employ first-order and third-order Markov predictors to see how the estimators perform with different predictors.

Figures 4 through 7 illustrate that 1-Marginal CBA and S1M reduce I/O time in comparison with VC. The one exception can be seen in Figure 5, when processing time is 20 ms, making the prefetch horizon 1, in which case the performances of the three estimators are virtually identical (< 0.2% difference between any two estimators on any of three traces). Figures 6 and 7 demonstrate that our estimators provide improvements independent of cache size and predictor order. Figure 8 shows that as disk access time increases the performance improvement increases. It increases in both absolute terms and percent difference. As the disk access time increases, the prefetch horizon

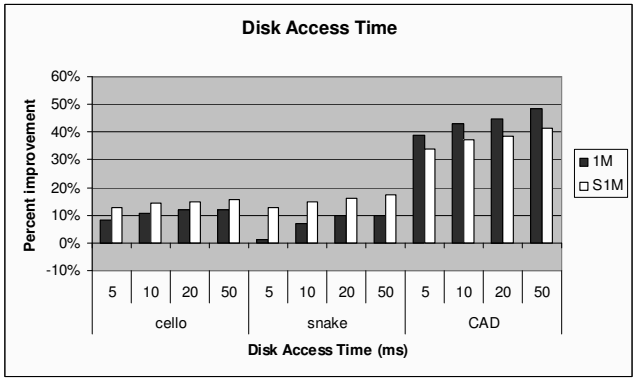

Figure 4.

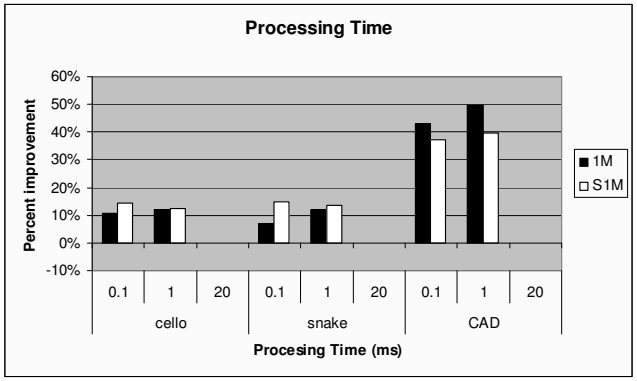

Figure 5.

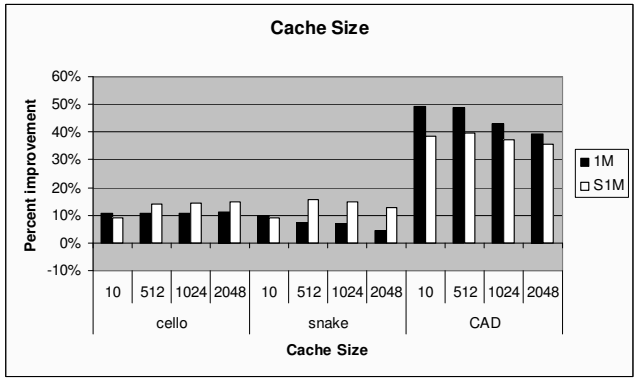

Figure 6.

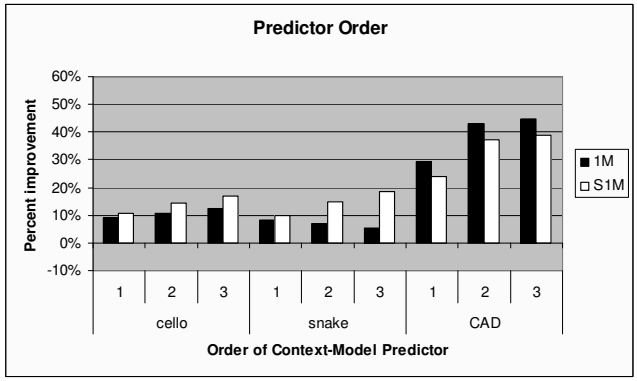

Figure 7.

increases, indicating that 1-Marginal and S1M work well when dealing with deep prefetches. Compared to VC, 1-Marginal offered an average reduction in I/O time of 19.1%, with a maximum of 49.8%. S1M offered an average reduction in I/O time of 20.1% with a maximum of 41.2%.

On the CAD trace, 1-Marginal performs better than S1M, but on the cello and snake traces S1M outperforms 1-Marginal. Though our theoretical model is based on a perfect predictor, the context-model used in the simulator is not perfect. When the actual success rate of prefetches is much higher than expected, then in general it would have been better to prefetch more. When the success rate is less than expected, in general it would have been better to prefetch less. For the traces we used, 1-Marginal prefetched many more blocks than S1M did. On all three traces, the success rate was less than the expected success rate, but on the cello and snake traces the success rate was much less than expected. This explains why S1M performed better than 1-Marginal even though S1M is an approximation of 1-Marginal. We have been working under the assumption of zero-cost analysis, but this is not a valid assumption in practice. Extra prefetches will incur not only the T_{driver} overhead, but also time to decide whether or not to prefetch them. Because S1M compares favorably to 1-Marginal and does so with fewer prefetches, S1M is the better candidate for practical implementations.

8. CONCLUSIONS

We have presented Opt, an algorithm for determining probabilistically optimal prefetching actions when provided with a finite probability tree. Though intractable in practice, it is of theoretical value. We have also demonstrated that any probabilistically optimal algorithm under this model must in some cases include one or more leaf nodes of the probability tree in its analysis to make a correct decision.

We introduced 1-Marginal and Simplified 1-Marginal prefetch estimators for prefetching in the presence of uncertainty. Simulations indicate that these estimators reduce I/O time better than existing techniques. This is particularly true when the prefetch horizon is large. (As the gap between processor speeds and disk/network latencies increases, the prefetch horizon grows.)

Our combined results offer more insight into the benefits (1M and S1M performance) and limitations (optimality requirements) of CBA. Although 1M and S1M now become the best CBA algorithms published to date, there needs to be continued work on making their implementations more efficient.

9. ACKNOWLEDGMENTS

Our thanks to Vivekanand Vellanki for providing access to the traces used in this study and for simulator code.

10. REFERENCES

[1] Albers, Susanne, Naveen Garg and Stefano Leonardi. Minimizing Stall Time in Single and Parallel Disk Systems, *Proc. 30th Annual ACM Symposium on Theory of Computing*, pages 454-462, 1998.

[2] Bartels, Gretta, Anna Karlin, Henry Levy, Geoffrey Voelker, Darrell Anderson and Jeffrey Chase. Potentials and

Limitations of Fault-Based Markov Prefetching for Virtual Memory Pages. Extended abstract, ACM SIGMETRICS, Atlanta, GA, May 1999.

[3] Bell, Timothy C., John G. Cleary and Ian H. Witten. *Text Compression*. Prentice Hall Adv Ref. Series, 1990.

[4] Cao, Pei, Edward W. Felten, Anna R. Karlin and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311-343, November 1996.

[5] Curewitz, Kenneth M., P. Krishnan and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 257-266, May 1993.

[6] Griffioen, James and Randy Appleton. Reducing File System Latency using a Predictive Approach. *Proceedings of the 1994 USENIX Summer Technical Conference*, June 1994.

[7] Highley, Timothy, Paul F. Reynolds, Jr. and Vivekanand Vellanki. Absolute Cost-Benefit Analysis for Predictive File Prefetching. University of Virginia Computer Science Department Technical Report CS-2002-11, April 2002.

[8] Kim, Jong Min, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, Chong Sang Kim. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. Technical Report No. SNU-CE-TR-2000-1, Seoul National University, May, 2000.

[9] Krishnan, P. and J.S. Vitter. Optimal Prediction for Prefetching in the Worst Case. *SIAM Journal on Computing*, Vol. 27, No. 6, pp. 1617-1636, December 1998.

[10] Kroeger, Thomas M. and Darrell D. E. Long. The Case for Efficient File Access Pattern Modeling. *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, IEEE, March 1999.

[11] Kroeger, Thomas M. and Darrell D. E. Long. Predicting File System Actions from Prior Events. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

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

[13] Madhyastha, Tara M. and Daniel A. Reed. Input/Output Access Pattern Classification Using Hidden Markov Models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57-67, San Jose, CA, November 1997.

[14] Ng, R., C. Faloutsos and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. *Proceedings of the 1991 ACM Conference on Management of Data (SIGMOD)*, pp. 387-396, 1991.

[15] Patterson, R. Hugo, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 19-95, 1995.

[16] Ruemmler, Chris and John Wilkes. UNIX disk access patterns. *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 405-420, January 1993.

[17] Thiebaut, Dominique, Harold S. Stone and Joel L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, Vol. 41, No. 6. June, 1992

[18] Vellanki, Vivekanand and Ann L. Chervenak. A Cost-Benefit Scheme for High Performance Predictive Prefetching. *Proceedings of Supercomputing 99*, November 1999.

[19] Vitter, J.S. and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, Vol. 43, No. 5, pp. 771-793, September 1996.

Appendix: Prediction Algorithm for Simulator of a File Prefetcher

We used a simulator to evaluate the performance of the 1-Marginal and S1M cost-benefit analysis estimators for file prefetching. Inaccurate predictions affect the ability to evaluate the quality of the estimators. We elected to use a context-model [5] for the predictor in our simulator.

INTRODUCTION

In a file prefetching system, there must be a mechanism to predict future accesses and a mechanism to establish when/whether to prefetch, based on those predictions. CBA addresses when and whether to prefetch. CBA estimators are formulae that express the benefit of fetching a block into the file cache or the cost of ejecting a block. The predictor is more important to performance than the estimator: it is possible to obtain significant speedup with a good predictor and naïve estimators, but excellent estimators will be of little use if the predictor has major flaws. However, as we have shown, if the predictor is good, different estimators can have a considerable impact on performance.

In order to best evaluate the impact of the estimators, it is important to have a predictor that is relatively accurate. With a predictor of lesser quality, it can appear that the better estimator is the one that prefetches more (if the prefetch success rate is higher than expected) or prefetches less (if the prefetch success rate is lower than expected), regardless of which estimator would perform better with accurate predictions. We sought a predictor that would enable us to evaluate the newly developed estimators. Here we describe the predictor we used and our rationale for its selection.

AVAILABLE APPROACHES

Several prediction algorithms are presented in the literature. Our main criteria for selecting a predictor were prediction accuracy and ease of implementation. We rejected some prediction algorithms because the predictions were based on whole files rather than file blocks [12]. Any of several other prediction algorithms may have met our needs [5][6][10][13].

Much prefetching work has dealt with prefetching deeply along a known access sequence [1][4], or making predictions for only the next access, if perfect knowledge of future accesses is not assumed. Since we wanted to prefetch deeply, we needed a prediction method that would provide predictions beyond the next access. Our work is based on a model very similar to that used in [18]. The prediction method utilized there is LZ [5]. The natural implementation for the LZ predictor is a tree, making deep predictions easy to identify. Additionally, LZ is optimal in the limit, though it converges slowly.

In our initial experiments, we used LZ but observed the problem described earlier: the estimator that appeared to perform best either prefetched more (if the prefetch success rate was higher than expected) or less (if the prefetch success rate was lower than expected), with little regard for our theoretical findings. We sought a better predictor and settled on the context-model predictor, also described in [5] for reasons described below.

PREDICTOR DESCRIPTION

Context-Models

Context-models are also known as Markov predictors. The *order* of a context-model is the length of the sequence that is used to determine the current context. For example, a third-order context-model for file prefetching will use the three most recent accesses to determine the current context, and make predictions based on what accesses previously occurred in that context.

Our specific predictor is prediction-by-partial-match (PPM) described in [5]. "A PPM prefetcher of order *m* maintains *j*th-order Markov predictors (on the page access sequence seen till now) for all $0 \leq j \leq m$." Our default predictor is of order 2.

Escape Probabilities

In [5], a fixed number of blocks were prefetched between any two accesses, with preference given to blocks predicted by higher order contexts. With this method, the context-models of different orders did not need to interact. For our approach, however, we needed to be sure that the probabilities of all predictions at any given time (from context-models of different orders) would not sum to more than one. The method to do this is called blending, and is described in detail in [3], p. 142.

Briefly, blending reduces the probabilities of the predictions of lower-order contexts, so that the total of all predictions is less than or equal to one. Escape probabilities are the probabilities of "escaping" to lower-order contexts. For instance, assume the escape probability at the highest-order context is 10%. Then the sum of the probabilities of the blocks predicted by the highest-order will be 90%, while the sum of the probabilities of the blocks predicted by all lower-order contexts will be 10%. There are a number of ways to assign escape probabilities, none of which is provably better than all others. We chose method A from [3], p. 144.

When the same block is predicted by more than one context, a full blending method will combine the predictions to produce a blended probability. Exclusions and lazy exclusions are methods to avoid that complication for faster, but possibly less accurate predictions [3]. We employ lazy exclusions.

Chaining

PPM is designed to make predictions for the next access only. In order to perform deep prefetches, it is necessary to have predictions for accesses further in the future. In order to accomplish this with PPM, we chained the predictions. For instance, if the most recent accesses were A and B, then AB would be the current context. If block C would be predicted in that context and then prefetched, then predictions for what

blocks might follow C (if it is accessed) would need to be determined. To do so, the BC context would be utilized, even though block C had not been accessed yet. Similarly, if the BC context were to predict block D and block D were to be prefetched, then the CD context would be used to predict other blocks. We call this chaining because on one round of prefetching decisions, several different contexts might be used for predicting, but they will all be linked, as AB is linked to BC, which is linked to CD.

EFFICIENCY

For efficiency in execution time, we restricted the number of blocks the predictor would predict. Two tunable parameters were introduced to facilitate this. The first, which we call *x*, determined the maximum number of blocks predicted at depth one. Priority was given to predictions coming from higher-order contexts. For instance, if x were twelve and the highest-order context predicted nine blocks, then up to three blocks would be predicted by the next highest-order context (with appropriate adjustments based on escape probabilities). We set x to twelve for the experiments in this paper.

The second parameter, which we call *y*, determined the maximum number of children predicted per node at depths greater than one. Again, priority was given to predictions coming from higher-order contexts. We set y to one for the experiments in this paper. We set y to one because high probability, deep prefetch candidates are likely to all be along the same branch, and because setting y greater than one has the potential to exponentially increase the number of blocks considered for prefetching, greatly slowing the simulation. With x set to twelve and y to one, there would be prefetching along twelve branches at most, but the prefetches could be to any depth (up to the prefetch horizon).

CONCLUSION

The predictor we have described scores well on our criteria of prediction accuracy and ease of implementation. Additionally, its execution speed is reasonable. We do not claim that the predictor is the best possible predictor on any single criterion, or even that it is the best choice based on some combination, but it performed well and enabled us to adequately evaluate our cost-benefit estimators.