Absolute Cost-Benefit Analysis for Predictive File Prefetching

Timothy Highley, Paul F. Reynolds
University of Virginia
{tjhighley, reynolds}@virginia.edu

Vivekanand Vellanki
IBM, Research Triangle Park
vellanki@us.ibm.com

Abstract

Cost-benefit analysis attempts to balance the benefits of file prefetching against the costs of giving up a buffer from the cache. This is a comprehensive approach to prefetching. It simultaneously answers the questions of what and when/whether to prefetch, and also addresses the interaction between prefetching and caching. Original research on cost-benefit analysis relied on hints about prefetching that were supplied by the program, though other research has considered the more general case of programs without hints. In this paper, we revisit previous research on cost-benefit analysis without hints, present new estimators for the costs and benefits of prefetching under a system model without hints, and generalize the system model to consider probability trees that may have the same block lie along multiple paths of the tree. We present a distinction between absolute and marginal cost-benefit analysis, two different approaches to file prefetching that previous researchers have used. The difference between the two is a subtlety that has not been previously noted or investigated. Our new estimators are based on absolute cost-benefit analysis, with modifications to address observed weaknesses in the absolute approach.

1 Introduction

File prefetching is a method to reduce the effect of disk and/or network latency on program execution time. The reduction is accomplished by requesting files before they are actually needed by the program. The gaps between processor speeds, memory speeds and disk speeds continue to increase, and it has become increasingly beneficial to prefetch files from the disk in order to alleviate disk latency. File prefetching is particularly advantageous when disk access costs are high, for example in many on-line programs.

As researchers have analyzed file prefetching, they have developed algorithms far more complex than one-block lookahead. Cost-benefit analysis, introduced by Patterson [Pat95], is one approach that has been shown to be quite successful. A cost is incurred when a buffer is re-allocated to hold a different block. The cost is weighed against the benefit (reduction in service time) that the program would receive from using the buffer for the new purpose. Cost-benefit analysis is very promising because it attempts to optimize the utilization of every single buffer. If every single buffer makes its maximum possible reduction in I/O service time, then the system overall will experience the greatest reduction in I/O service time.

Patterson's approach was originally applied to systems that depend on hints, but has been extended to those that do not [Vel99]. This extension is important because not all programs provide hints, and in some cases it is not feasible to have the program supply hints. Without hints, the prefetch decisions must be probabilistic, and the possibility of mispredictions becomes a serious concern because a misprediction takes a buffer away from the LRU cache. The cost-benefit approach seeks to find the correct balance so that the more tentative predictions will not be prefetched and harm performance but the more reliable predictions will be prefetched.

In this paper, we revisit the work from [Vel99] and present new estimators for the system model where file access hints are not provided by the program. We also generalize the system model to consider probability trees that may have the same block lying along multiple paths. We explain the difference

between absolute and marginal cost-benefit analysis, which are two different approaches to cost-benefit analysis that previous researchers have employed without noted appreciation for the differences in the approaches. We base our estimators on the absolute approach, but introduce modifications to address limitations that the absolute approach exhibits, calling the modified algorithm quasi-absolute.

2 Cost-Benefit Analysis

The system model of Patterson, et al [Pat95] assumes hinting programs. Programs give hints to disclose some of the blocks that they will access. It is guaranteed that if a hint is given for a particular block, then the block will indeed be accessed. Between hinted blocks, there may be demand (unhinted) accesses. Patterson assumes that there is no congestion at the disk (i.e. infinite disk parallelism).

The file cache is divided into two portions, the hinted cache and the LRU cache. The hinted cache contains file blocks that have been prefetched but not yet accessed. The LRU cache is just that: LRU. Conceptually, a file block may be part of both portions of the cache at the same time, if it is one of the most recently used blocks and a hint has also been issued that indicates the block will be accessed again in the future. (If it is conceptually in both portions of the cache, it only occupies one physical buffer.) Four estimators express the cost of ejecting a block from the LRU cache, the cost of ejecting a block from the hinted cache, the benefit of prefetching a block and the benefit of fetching a demand request. The estimators are expressed in terms of the *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, a fetch is initiated.

Vellanki and Chervenak [Vel99] base their model on the Patterson model, however they do not assume hinting processes. Instead, they assume a tree representing tallied accesses of the process. The past history is then used as a predictor of future accesses. This alternative has an effect on the estimators, and Vellanki derives new values for those estimators. We explore the no-hints system model, and present new results.

2.1 System model

Our system model is essentially the same as that presented in [Vel99]. We assume a program consisting of $\mathbf{N_{I/O}}$ file accesses, with a constant amount of processing time between accesses, $\mathbf{T_{CPU}}$. The time needed to satisfy a file access from the file 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 $\mathbf{T_{disk}}$ to wait for the fetch to be completed. $\mathbf{T_{stall}}$ is the time the processor spends waiting for the file block; it is equal to T_{disk} if the block is not prefetched or cached, but it may be less if it is prefetched or cached.

We assume the existence of a probability tree that dynamically represents sequences of potential future accesses. We associate block names with nodes in the tree. For now we assume that a block will have only one node. We do not yet consider the question of what to do if a block appears at multiple positions in the probability tree, but we will address that question in Section 2.4.3.1. We assume the tree is accurate but incomplete. By accurate, we mean that when an edge is labeled with a probability, that probability corresponds to the actual probability of following the edge at that time. We assume this accuracy for simplicity of analysis. In a real system, such probabilities will be estimates with some degree of error. The impact of the inaccuracy of the probability tree is left as an open problem. By incomplete, we mean that the sum of the probabilities on the outbound edges of a given node do not necessarily sum to one, though the sums are always less than or equal to one, as the accuracy condition implies. A node is a *candidate node* if the blocks of all of the node's ancestors have already been prefetched. A block is a *candidate block* if it has a candidate node. (The distinction between candidate nodes and candidate blocks is only relevant in the replicated case introduced in Section 2.4.3.2.) A block,

b, has two major attributes: its depth $\mathbf{d_b}$, which is the number of edges that must be followed to reach the block's node in the tree; and its probability $\mathbf{p_b}$, which is the probability of following the path to the node (i.e. the product of the probabilities associated with edges between the root of the tree and block b's node).

An access period consists of a prefetch decision (which we assume to be zero-cost), T_{driver} for each prefetch, and then T_{stall} (if any), T_{hit} and T_{CPU} for the current access. After T_{CPU}, there is another file access and then the next access period begins.

If an access is made that is not reflected in the tree then the access sequence is following a path that could not have been predicted with any confidence. All pending prefetches were incorrect, and the probability tree occurring after the demand access may be very different from the one before the demand access.

2.2 Buffer costs and prefetching benefits

In [Vel99], an estimator is given for the benefit of prefetching a particular block b that has parent block x. $\mathbf{\Delta T_{pf}(b, d_b)}$ is defined as the amount of time saved by prefetching block b at depth d_b, compared to the time required to fetch b on demand. If the probability of accessing parent block x is $\mathbf{p_x}$ then the benefit is

$$B(b) = p_b \Delta T_{pf}(b, d_b) - p_x \Delta T_{pf}(x, d_b - 1) \qquad (1)$$

We address two assumptions in the derivation. The first relates to buffer usage (or *bufferage*). The unit for bufferage is buffer-access, which is defined as the occupation of one buffer for one access-period for the purpose of a possible future access to the block. The buffer usage is taken to be one buffer-access in the above estimator. We argue the bufferage should actually be one buffer *per access period*. The buffer will likely be held for more than one access period, so the bufferage is more than just one buffer-access. Second, the second term of equation 1 is meant to indicate that part of the benefit of prefetching block b is already realized by prefetching the predecessor, block x. This is true, in a sense, because block x should always be prefetched before block b is prefetched. However, the benefit of prefetching block b is in the context of all of b's ancestor blocks and not only dependent on its immediate parent block.

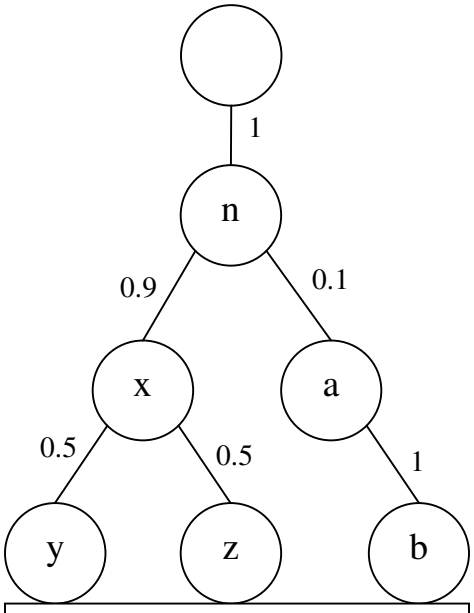

Figure 1:
Prefetch block b or block y?

Consider the following example (see Figure 1). Assume there are six file blocks: n, x, y, z, a and b. The next three blocks accessed will be either (n, x, y) with 45% likelihood, (n, x, z) with 45% likelihood or (n, a, b) with 10% likelihood. Then the probabilities of accessing the individual blocks are $p_n = 1$, $p_a = 0.10$, $p_b = 0.10$, $p_x = 0.90$, $p_y = 0.45$ and $p_z = 0.45$. Assume that blocks n, a, x and z are already cached and there is one additional cache buffer available for prefetching. We must then decide to prefetch either block y or block b. Clearly, block y is more than four times more likely to be accessed than block b, and it is difficult to imagine why block b should be prefetched instead of block y, but equation 1 determines that block b should be prefetched. In that approach, a block is very likely to be taken if the probability of accessing the block is close to the probability of accessing its predecessor, even if that probability is very low. (Notice that $p_a = p_b$ but $p_x > p_y$.) Even when the prefetching overhead is taken into account, to compensate for the penalty of prefetching blocks that are not accessed, the choice of block b over block y is not changed in [Vel99].

These observations led us to revisit this system model. We describe interesting results, as we first draw a distinction between absolute and marginal cost-benefit analysis, described in Section 2.3, and then present new estimators in Section 2.4.

2.3 Absolute vs. marginal cost-benefit analysis

Patterson [Pat95] introduced the concept of common currency, which is defined as "the magnitude of the change in I/O service time per buffer-access". This concept permits a comparison between two potential prefetch blocks when they are expected to occupy a buffer for different amounts of time. When an estimator is expressed in the common currency, the numerator is the expected change in I/O service time and the denominator is the expected change in bufferage. In both the numerator and denominator, "expected change" refers to the delta between two possible scenarios. The numerator and the denominator for a single estimator must be consistent and refer to the same two scenarios.

The two scenarios could be prefetching a block now and not prefetching the block at all. That is, the numerator could express the time saved by prefetching now versus not prefetching at all, and the denominator the extra bufferage needed to prefetch now versus not prefetching at all. That is not the approach Patterson takes in his cost estimator for ejecting from the prefetch cache. For that estimator, the first scenario is not ejecting the block. The second is ejecting a block now and then prefetching the block back later at some unknown time. The numerator, then, is the extra time required because the block was ejected (assuming it would be prefetched back later). The denominator is the bufferage freed by ejecting the block (assuming it would be prefetched back later). Both approaches just described are consistent, but they are derived in different ways. We refer to the first as *absolute cost-benefit analysis* and the second as *marginal cost-benefit analysis*.

In absolute cost-benefit analysis, no assumptions are made about future actions when calculating the estimators. That is, future fetches are not taken into account and it is assumed that any block that is not currently in the cache will be fetched on demand. In essence, absolute cost-benefit analysis compares the effect of using the cache for a certain purpose and the effect of not using the cache at all.

In marginal cost-benefit analysis, guesses or assumptions are made about future fetches in order to possibly take actions that are more time critical. This is Patterson's approach in his cost estimator for ejecting from the prefetch cache. When considering whether or not to eject a block in this case, it is known that the block will have to be fetched back at some point and so it makes sense to take into consideration the fact that the ejected block may not stall for the entire T_{disk}. In essence, marginal cost-benefit analysis compares the effect of taking an action and the effect of taking the action at a later point in time.

Marginal cost-benefit analysis is a broad term, referring to any approach that compares between using the cache now and using it later. We note two special types of marginal cost-benefit analysis. If the two scenarios in question are using the cache now and using it at the next opportunity, we call this *pure* marginal cost-benefit analysis, since it is considering the smallest margin for comparison. If the two scenarios being compared are taking the action now and taking the action at the "best" future time, we call this *optimistic* marginal cost-benefit analysis. For example, Patterson's simulator handles the cost estimator for ejecting from the prefetch cache with an optimistic approach. It assumes that an ejected block will be prefetched back just in time to suffer no stall.

There are cases where using marginal cost-benefit analysis can lead to good decisions that absolute cost-benefit analysis would not make. For instance, if a block can be ejected and then refetched and still not suffer a stall, then the buffer could be used for another purpose for a period of time. If the block is sure to be accessed eventually, then absolute cost-benefit analysis may give a high value to retaining that block and not allow the buffer to be used for another purpose.

There are also cases where use of marginal cost-benefit analysis can incur risks that do not pay off. Certain amount of guessing about future actions is incorporated in marginal cost-benefit analysis, and if those guesses are incorrect then the strategy represented by marginal cost-benefit analysis may perform worse than the absolute cost-benefit analysis approach. Figure 2 presents an example where use of optimistic marginal and absolute CBA leads to different decisions. Using the optimistic marginal CBA method would result in a prefetch while using the absolute CBA approach would not.

We focus first on absolute cost-benefit analysis. When constructing his own approach, Patterson recognized that it is difficult to dynamically compute when a block would be refetched. We agree. Furthermore, our system model does not assume a hinted stream of accesses, further complicating the already difficult issue of guessing about future prefetch actions.

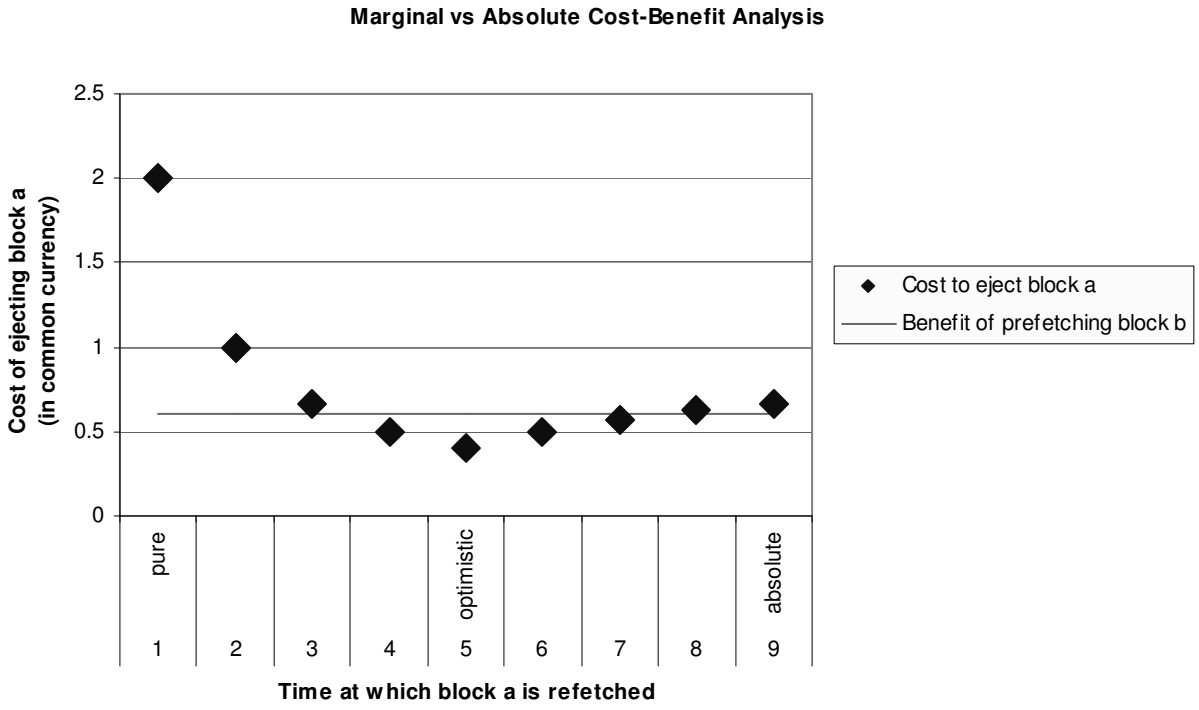

Figure 2: Assume that block a, which is in the cache, will be accessed 9 time units in the future; that block b, if it is accessed, will be accessed 5 time units in the future; and that the benefit of prefetching block b is 3/5. That is, the expected time savings is 3 time units and expected bufferage is 5 buffer-accesses. *Absolute CBA*: We assume that block a will not be refetched until it is needed, so the cost of ejecting it is 6/9. (A buffer is freed for 9 time units, and 6 time units will be added to the total I/O time, 2 units for T_{driver} and 4 units for $T_{disk} + T_{hit}$). In this case, block b will not be prefetched in place of block a because the cost is greater than the benefit. *Optimistic marginal CBA*: We assume block a will be refetched at t_5, so the cost of ejection is 2/5. (A buffer is freed for 5 time units, and 2 time units for T_{driver} are added to the total I/O time. Block a is refetched early enough to avoid any stall.) Here, block b will be prefetched into block a's place because the benefit is greater than the cost. In this case, marginal CBA can save time over absolute CBA because neither block b nor block a experience stall if the marginal CBA guess is correct. However, if the marginal CBA guess was incorrect and block a is not able to be refetched until time 9, then the marginal CBA would perform worse than absolute CBA.

2.4 Deriving estimators

In this section we derive absolute cost-benefit estimators under the system model described in Section 2.1. The benefit for a demand fetch is infinite, as described in [Pat95].

2.4.1 Total$_{I/O}$ time: the metric

The total running time of a program is

$\mathbf{T}=N_{I/O}(T_{CPU} + \mathbf{T_{I/O}})$

$T_{I/O}$ is the average amount of time spent on I/O per file access. Our objective is to minimize the value T. We assume that the number of I/O accesses, $N_{I/O}$, and the amount of time spent computing between accesses, T_{CPU}, have predetermined values and we focus on the minimization of I/O time. The total amount of time the program spends on I/O is

$Total_{I/O} = (N_{I/O})(T_{I/O})$

This is the metric we seek to minimize. As Cao demonstrated [Cao96], it is impossible to always minimize the value for $T_{I/O}$ without perfect knowledge of the access sequence, so our approach is probabilistic.

2.4.2 Calculating bufferage

One important factor in figuring the cost or benefit of ejecting or prefetching a block is the amount of bufferage. Here we calculate the expected bufferage for any node in the probability tree.

If block b occupies a buffer (or is being prefetched) then we define $\mathbf{buf_b(d)}$ as the expected amount of bufferage block b will use when its node in the tree is at depth d. Block b will release the buffer either after the block is accessed or after it is determined that the block will not be accessed. To calculate $buf_b(d)$, mark block b's node and mark each node n if n is not an ancestor of block b's node, but n's parent is an ancestor of block b's node. (Figure 3 illustrates the marked nodes for a more general case.)

$buf_b(d) = \sum_{n = \text{marked nodes}} d_n p_n$

2.4.3 Selecting which block to prefetch

In choosing a block, we apply a greedy approach to minimizing the value for $Total_{I/O}$. In order to select the best block we utilize the common currency.

One option that will always be available is to not prefetch anything. An unused buffer will not reduce the program running time, but neither will it increase the running time. In reality, the buffer will contain some other data, but even if the buffer were empty there would be zero effect on the I/O service time. (A misprediction, on the other hand, could actually harm performance due to the overhead of initiating the fetch.)

Prefetching decisions are made at the beginning of each access period. The expected benefit in the common currency is calculated for each block. This requires calculating the expected change in I/O service time divided by the expected time that the block will occupy a buffer. For each block, p_b is the probability that the edges that are followed in the tree lead to block b. In other words, p_b is the product of the probabilities of all the edges leading to block b's node from the current (root) node. Recall that d_b is the number of accesses in the future that block b would be accessed if the path would indeed be taken. That is, d_b is the depth in the tree of block b.

We have already calculated the expected bufferage in Section 2.4.2. For I/O time, there are two cases: if the prefetch is correct and if it is incorrect. For each case, the expected impact on I/O time needs to be determined.

If the prefetched block is not used, then T_{driver} overhead processing time is added to the program's total I/O time, as compared to the value if no prefetching was performed. There is a $(1-p_b)$ chance that when block b is prefetched it will not be accessed. If it is prefetched but not used then T_{driver} processing time is added to $Total_{I/O}$.

If the prefetched block is accessed, then the total amount of time spent on I/O in the program is reduced. If the block is prefetched at depth zero, the access will stall for the entire disk access time, T_{disk}. For simplicity, we assume T_{disk} is constant, in keeping with the assumptions of [Pat95] and [Vel99]. If the block is prefetched at a depth $d_b > 0$, then the processor will stall for less time. During each access period, the processor is busy processing (T_{CPU}), busy reading a hit from the disk cache (T_{hit}) and busy issuing, on average, **s** new prefetches (**s**T_{driver}). In addition, any stall that occurs before block b is accessed will mask part of the stall for block b. We must also bound T_{stall} below by zero, since negative stalls cannot occur. Thus, a block b that is prefetched at a depth d_b causes a stall of

$$T_{stall}(d_b) = \max (T_{disk} - [d_b(T_{CPU} + T_{hit} + sT_{driver}) + (\sum_{m = b\text{'s direct ancestors}} T_{stall}(m))], 0)$$

In the equation above, $T_{stall}(m)$ is zero if block m is cached, is T_{disk} if block m is neither cached nor being prefetched, and is the stall value that was calculated when block m was prefetched, if block m is currently being prefetched.

The expected change in stall time for block b (if it is accessed), is then

$$\Delta T_{stall}(b) = T_{stall}(d_b) - T_{disk} =$$

$$\max (T_{disk} - [d_b(T_{CPU} + T_{hit} + sT_{driver}) + (\sum_{m = b\text{'s direct ancestors}} T_{stall}(m))], 0) - T_{disk} =$$

$$\max (- d_b(T_{CPU} + T_{hit} + sT_{driver}) - (\sum_{m = b\text{'s direct ancestors}} T_{stall}(m)), -T_{disk})$$

The change in stall time is the only effect the prefetch will have on $Total_{I/O}$ if the block is accessed. Expressed in the common currency, the overall benefit of prefetching block b (called **B(b)**) is

$$B(b) = E[\mathbf{\Delta Total_{I/O}}] / E[\text{buffer-usage}] = \qquad (2)$$

$$\{p_b[\max (- d_b(T_{CPU} + T_{hit} + sT_{driver}) - (\sum_{m = b\text{'s direct ancestors}} T_{stall}(m)), -T_{disk})] + (1-p_b)[T_{driver}]\} / \{buf_b(d_b)\}$$

This equation expresses expected change in I/O service time divided by expected buffer usage. The block that produces the most negative value for B is the block that should be prefetched (if any block is prefetched). If no block produces a benefit greater than the benefit of retaining the blocks in the cache, then the option to fetch no block at all is the best option and therefore no block should be prefetched.

Note that $buf_b(0)$ is zero, indicating an undefined benefit for prefetching at depth zero. Since a prefetch at depth zero is equivalent to a demand fetch, the benefit should be infinite.

2.4.3.1 Anomaly for absolute cost-benefit analysis

In the previous section, we derived a benefit estimator for absolute cost-benefit analysis. Previous work has limited the pool of blocks under consideration to those that meet the definition of a candidate block: a block is a candidate block if every block in the tree that is a direct ancestor of that block has already been accessed or has already been prefetched. Is there any reason to assume this condition under the model of absolute cost-benefit analysis? The reason to limit the number of blocks under consideration is that a descendant block will only be accessed after its ancestor blocks. A prefetch for the descendant block can then be initiated after initiating the fetches for the ancestors, since all of the fetches for the ancestors must complete before the descendant is accessed. However, absolute cost-benefit analysis assumes that blocks

not in the cache will be fetched on demand. Restricting the pool of candidates implicitly assumes a condition that is only consistent with marginal cost-benefit analysis, since the motivation there is that the descendant can be prefetched later. Therefore, an absolute cost-benefit approach should not make the restriction.

If every block is always considered for prefetching, an interesting anomaly occurs. If there are two blocks under consideration and their relationship in the probability tree is parent-child, under absolute cost-benefit analysis it is more advantageous to prefetch the child block. If the child block is prefetched, all of its stall will be masked by the demand fetch of the parent block, but if the parent block is prefetched, only part of its stall will be masked (by the compute time that comes before it is accessed). In this case, the child block could also be prefetched while the parent block is being prefetched, but absolute cost-benefit analysis does not take this into consideration.

We conclude that some sort of marginal consideration needs to be included in order to make good prefetch decisions. We define the approach *quasi-absolute*, in which the only blocks under consideration are those that meet the definition of a candidate block.

2.4.3.2 Replication of a block within the probability tree

The benefit estimator expressed in equation 2 evaluates the benefit of a prefetch for each node of the probability tree, where each node represents a particular file block. However, a particular file block may appear at more than one node in the probability tree. We call these blocks *replicated* blocks. It may be the case that the best block to prefetch is one that appears along several paths in the probability tree, even though none of the paths is particularly likely to be followed. The estimator in equation 2 does not take this into account.

If a block occurs twice along the same path, then the block would be fetched into the cache at the first occurrence, and would then already be in the cache for the second occurrence. As Patterson noted [Pat95], "it only takes one disk I/O to fetch a block no matter how many times the block is accessed thereafter." For this reason, we ignore nodes for replicated blocks when the node has an ancestor node that is also for the same block.

Equation 2 includes three attributes of a potential block: p_b, d_b and $buf_b(d_b)$. In order to find the benefit estimator for prefetching a block that is replicated along multiple paths, we need to determine the bufferage and expected impact on I/O time if this block is prefetched.

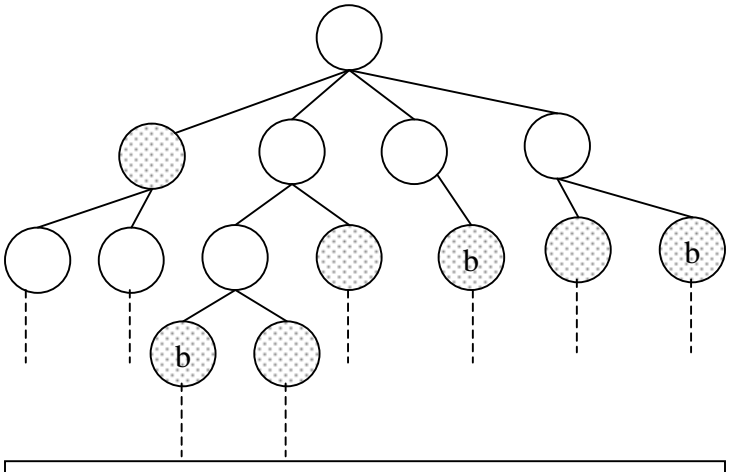

Figure 3: Shaded nodes indicate which ones to mark when calculating bufferage for a replicated block b

We consider the bufferage for a block that appears more than once in the tree, where the block's set of nodes is $N_b = \{n_{b1}, n_{b2}, \ldots\}$. The bufferage with multiple occurrences of a block in the tree is an extension of bufferage for the non-replicated case (Section 2.4.2). The buffer can be released either after the block is read or after it can be determined that the block was mispredicted. Mark each node n of the tree if it is either one of block b's nodes or the node does not have any of block b's nodes as descendants, but n's parent node does have a block b node as a descendant. Figure 3 illustrates which nodes should be marked. Then the expected bufferage for the replicated

block b is

$\mathbf{buf_b(N_b)} = \sum_{n = \text{marked nodes}} d_n p_n$

Note that the expected bufferage for multiple nodes is always greater than or equal to the expected bufferage for any subset of those nodes, because adding more nodes of the same block can only extend how long a buffer is held.

If a replicated block b is prefetched the expected change in I/O time is a weighted sum over the block's nodes, plus the potential overhead. This formula is derived by considering the probability of following each branch of the tree, and the impact on I/O time that is realized if that branch is followed. Finally, the impacts on I/O time for the different branches are summed.

$E[\Delta Total_{I/O}] = \{\sum_{n = \text{nodes for b}} p_n[\max (- d_n(T_{CPU} + T_{hit} + sT_{driver}) - (\sum_{m = \text{n's direct ancestors}} T_{stall}(m)), -T_{disk})]\} + (1 - \sum_{n = \text{nodes for b}} p_n)[T_{driver}]$

Adding instances of a block can only improve the impact on total I/O time. This is because when a node is added to the set the probability of a misprediction is being reduced and the probability of a correct and timesaving prefetch is being increased.

Define A(n) as the set of nodes that are direct ancestors of node n. The benefit of prefetching a block under absolute cost-benefit analysis, considering all of the nodes in the set N_b is

$\mathbf{B(N_b)} = E[\Delta Total_{I/O}] / buf_b(N_b) =$
$\{\{\sum_{n = Nb} p_n[\max (- d_n(T_{CPU} + T_{hit} + sT_{driver}) - (\sum_{m = A(n)} T_{stall}(m)), -T_{disk})]\} + (1 - \sum_{n = Nb} p_n)[T_{driver}]\} / \{buf_b(N_b)\}$

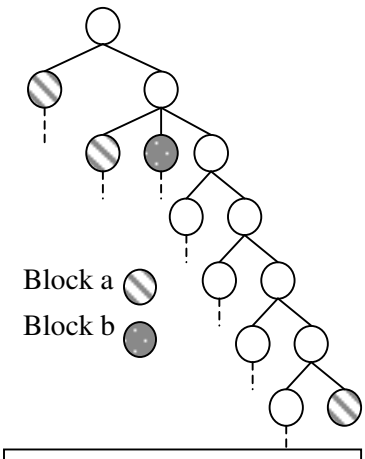

Figure 4: Block a appears in 3 nodes. If we consider the two nodes of block a nearest the root node, then $B(a_1, a_2) > B(b)$. If we also consider the third node at the bottom of the figure, then $B(a_1, a_2, a_3) < B(b)$, since a's third node increases the expected bufferage to the extent that it appears more beneficial to prefetch block b.

Though not immediately obvious, it may be the case that a candidate node at a deep depth but a low probability actually harms the overall benefit for the block. This can happen because including the block in the set might slightly improve the expected change in I/O time, but greatly increase the expected bufferage (see Figure 4). In order to handle this anomaly, we recognize that the block can be prefetched but then ejected early, even if the path that is followed leads toward the deep but low probability node. Essentially, we can ignore the deep but low probability node to reduce the expected bufferage. However, there may be other deep nodes that are beneficial, so a node should not be excluded based solely on its depth in the probability tree.

With this observation, the benefit estimator for prefetching block b is modified so that some of the nodes can be selectively excluded from consideration. The estimator is then the maximum value of B(J) where J is any subset of N_b:

$\mathbf{B(b)} = \max \{B(J) \text{ where } J \subseteq N_b\}$

The anomaly noted in Section 2.4.3.1 also applies to the replicated case. If we include any non-candidate nodes in a set N_b, then absolute cost-benefit analysis will heavily favor them. The *quasi-absolute* approach for the replicated case, therefore, excludes the non-candidate nodes from consideration. Block b's non-candidate nodes should actually have some positive impact on the benefit estimator, but not to the extent that absolute cost-benefit analysis gives them.

2.4.4 Benefit of retaining a block already in the cache

[Pat95] and [Vel99] balanced costs against benefits. The cost of ejecting a block is essentially the loss of the benefit that would have been experienced if the block was retained. Since the cost of ejection is the same as the benefit of retention, it is valid to speak in terms of "benefit vs. benefit" rather than "cost vs. benefit". We prefer the former in our analytic model because then the benefit of each block can be ranked and those blocks with the best benefits can be selected to be in the cache, regardless of whether they are already in a buffer or not. The benefit is simply calculated differently if the block is already in the cache.

If a block already in the cache is accessed, then the total I/O time is reduced by ($T_{disk} + T_{driver}$), relative to the total I/O time if there was no cache. This is true whether the block is part of the LRU list, appears in the probability tree or both. In this section, we only consider blocks that appear in the probability tree. In Section 2.4.4.1 we describe how the LRU list can be reflected in the probability tree.

The benefit of retaining a block is the expected change in I/O service time divided by the expected change in buffer usage. For a block that appears in only a single node, the bufferage is the same as derived earlier and the expected change in I/O time is

$E[\Delta Total_{I/O}] = p_b[-(T_{disk} + T_{driver})]$

If we are considering the benefit of retaining a block that is replicated in the tree in the nodes $N_b = \{n_{b1}, n_{b2}, \ldots\}$, the situation is very similar to Section 2.4.3.1.

$E[\Delta Total_{I/O}] = (\sum_{n=Nb} p_n)[-(T_{disk} + T_{driver})]$

The bufferage and overall benefit of retaining a replicated block is calculated as described in Section 2.4.3.1. For each block b that has a set of nodes N_b, the benefit of retaining block b, considering all of its nodes, is

$\mathbf{B(N_b)} = E[\Delta Total_{I/O}] / buf_b(N_b) =$

$(\sum_{n=Nb} p_n)[-(T_{disk} + T_{driver})] / buf_b(N_b)$

The anomaly with deep, low probability nodes still exists, so the benefit of retaining block b is the maximum value of B(J) where J is any subset of N_b. There is no need to restrict the blocks considered for retention to only the candidate blocks, since all retained blocks experience zero stall. The absolute and quasi-absolute approaches have the same estimator for the benefit of retaining a block.

$\mathbf{B(b)} = \max\{B(J) \text{ where } J \subseteq N_b\}$

2.4.4.1 The effect of the LRU cache

In our system model, we have assumed that at any point we have an accurate but incomplete probability tree that reflects potential future accesses. Up to this point, we have not addressed the list of most recent file accesses because it is possible to have the list reflected in the probability tree. For our analytic model we assume the existence of the probability tree and zero-cost analysis. Because we assume zero-cost analysis, we can take the time to do a transformation so that the LRU list is reflected in the probability tree. We now describe how to have the LRU list reflected in the probability tree.

As described in [Pat95], the probability that the nth least recently used block will be accessed is H(n) – H(n-1), where H(x) is the demand cache-hit ratio when x buffers are in use in the demand cache. Since both the block and the probability of it being accessed are known, an explicit node in the tree for this block can be added, with the appropriate probability. If the block was already reflected in the tree due to a prediction, then the probabilities can simply be summed. In this way, the information from the LRU list is reflected in the probability tree and the LRU list does not need to be included as part of the analytic model. We recognize that for simulations and real-world performance, the LRU list is an important factor, but our focus here is on making the best selection based on zero-cost analysis and an accurate but

incomplete probability tree. In this model, the LRU list does not need to be taken into consideration explicitly because its impact can be completely reflected in the assumed probability tree.

2.5 Integrated algorithm

We have developed two estimators under absolute cost-benefit analysis – one for fetching a block and one for retaining a block. The estimator for retaining a block addresses any block that is in the file cache, regardless of its initial reason for being fetched. The estimator for fetching addresses prefetches, but implicitly handles demand requests as prefetches of depth zero. The integrated algorithm computes for each block under consideration, after each access request, the benefit of fetching the block or retaining the block. Those blocks with the lowest benefit of retention will be ejected in favor of the blocks with the highest benefit of fetching, so long as the benefit of the fetch is higher than the benefit of retention.

The quasi-absolute algorithm restricts the set of blocks considered for prefetching to those blocks that meet the definition of a candidate block.

3 Related work

Prefetching remains an area of active research, though much of the recent research in prefetching has moved away from typical file prefetching to web prefetching [Mal99][Ibr00]. Some recent work in file prefetching includes [Kri94], [Alb98] and [Cao96]. These papers describe research on prefetching problems where the entire access sequence is known in advance.

Other researchers have addressed the problem of predicting future file accesses. The idea of compiler-inserted prefetching hints has been presented by Mowry [Mow96]. Chang transforms binaries so that they will have a speculative thread. The speculative thread runs ahead to see what the future accesses are and issues (possibly incorrect) hints [Cha99].

Many other approaches to predicting future file accesses fall into the category of history based prefetching methods. The various history-based approaches can be distinguished from one another in a number of ways.. The various approaches assume differing amounts of historical knowledge. They also have different interpretations about what “current context” means. For example, some are whole file prefetching schemes, and the context is determined by the sequence of files that are touched, while others are file block prefetching schemes. The most significant differences arise in how they interpret the data to decide exactly what to prefetch.

Some of the history based methods include the Markov predictor of [Bar99], the data compression predictors (LZ, PPM, FOM) of [Cur93], hidden Markov models [Mad97], partitioned context model [Kro96], the path-based target prediction scheme presented in [Whi00], the analytic approach of [Lei97], and the graph-based approach of [Gri94].

Krishnan and Vitter use the LZ data compression predictor, and then make a randomization of what to prefetch in order to optimize for the worst-case scenario [Kri94]. This is because a random algorithm is necessary to optimize for the worst case [Cov67].

As we have mentioned throughout the paper, the work closest to our own is Patterson’s initial work on cost-benefit analysis [Pat95] and Vellanki’s previous work in the area of predictive cost-benefit analysis [Vel99]. Tomkins investigated cost-benefit analysis without assuming infinite disk parallelism, but maintaining the assumption of a hinted stream [Tom97].

4 Future work

We have investigated absolute cost-benefit analysis and identified weaknesses that demonstrate the need for marginal considerations. We addressed the weaknesses by placing restrictions on the set of blocks under consideration, but we plan to more fully explore the potential of marginal cost-benefit analysis.

The work we have presented here is based on a particular system model. In the future we will generalize this approach to system models that do not assume infinite disk parallelism and do not assume a constant time between file accesses. We will also explore the impact of attempting to impose a maximum stall time on the system in order to provide a more consistent level of service (even if the overall program time increases). We will address the practical implications of our results at greater length in future work.

5 Conclusion

In this paper, we drew a distinction between two approaches to cost-benefit analysis that have appeared in the literature previously, and termed those approaches absolute and marginal cost-benefit analysis. We presented absolute cost-benefit estimators under a specific system model and demonstrated weaknesses in the estimators. We then presented quasi-absolute, a cost-benefit approach that is based on absolute cost-benefit analysis principles but addresses the weaknesses we noted. Quasi-absolute does have limitations. In particular, it does not take into consideration the effect of non-candidate nodes of replicated blocks. Our work has demonstrated insights into making cost-benefit decisions and paved the way for thorough investigation of marginal cost-benefit analysis.

Our system model is similar to that of [Vel99] because we are using a predictive approach rather than relying on hints. We presented an improvement to the concept of bufferage presented in that paper, and at the same time analyzed a more general system that takes into account the situation when the same block lies along multiple paths of the probability tree. Finally, we described how, for our predictive analytic model, the LRU list does not need separate analysis since it can be completely represented in the probability tree.

6 References

[Alb98] Albers, Susanne, Naveen Garg and Stefano Leonardi. Minimizing stall time in single and parallel disk systems, *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 454-462, 1998.

[Bar99] 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.

[Cao96] 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), pp. 311-343, November 1996.

[Cha99] Chang, Fay and Garth Gibson. Automatic I/O hint generation through speculative execution. *Proceedings of 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pp. 1-14, New Orleans, LA, February 1999.

[Cov67] Cover, T.M. Behavior of predictors of binary sequences. *Proceedings of the 4th Prague Conference on Information Theory, Statistical Decision Functions, Random Processes*, Publishing House of the Czechoslovak Academy of Sciences, pp. 263-272, Prague, 1967.

[Cur93] Curewitz, Kenneth M., P. Krishnan and Jeffrey Scott Vitter. Practical prefetching via data compression. *Proceedings 1993 ACM-SIGMOD Conference on Management of Data*, pp. 257-266, May 1993.

[Gri94] Griffioen, James and Randy Appleton. Reducing file system latency using a predictive approach. *Proceedings of the 1994 USENIX Summer Technical Conference*, June 1994.

[Iba98] Ibanez, P, V. Vinals, J. Briz and M. Garzaran. Characterization and improvement of load/store cache-based prefetching. *Proceedings of the 12th International Conference on Supercomputing,* pp. 369-378, July 1998.

[Kri94] Krishnan, P. and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6), pp. 1617-1636, December 1998. Another version appeared in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 392-401, January 1994.

[Lei97] Lei, Hui and Dan Duchamp. An analytical approach to file prefetching. *Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 275-288, January 1997.

[Mad97] Madhyastha, Tara M. and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 57-67, San Jose, CA, November 1997.

[Mal99] Maltzahn, Carlos, Kathy J. Richardson, Dick Grunwald and James Martin. On bandwidth smoothing. *Proceedings of the 4th International Web Caching Workshop*, San Diego, CA, March 1999.

[Mow96] Mowry, Todd C., Angela K. Demke and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. *Second Symposium on Operating System Design and Implementation*, pp. 3-17, Seattle, WA, October 1996.

[Pat95] Patterson, R. Hugo, Garth A. Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka. Informed prefetching and caching. *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 19-95, 1995.

[Tom97] Tomkins, Andrew, R. Hugo Patterson and Garth Gibson. Informed multi-process prefetching and caching. *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems* (SIGMETRICS), June 1997.

[Vel99] Vellanki, Vivekanand and Ann L. Chervenak. A Cost-Benefit Scheme for High Performance Predictive Prefetching. *Proceedings of Supercomputing 99*, November 1999.

[Whi00] White, Brian and Kevin Skadron. Path-based target prediction for file system prefetching. Technical report CS-2000-06, University of Virginia Department of Computer Science, February 2000.