Interleaving sequences to maximize the minimum prefix sum

Tracy Kimbrel

Technical Report 96-12-01

Department of Computer Science and Engineering
University of Washington

Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA

Interleaving sequences to maximize the minimum prefix sum

Tracy Kimbrel
Dept. of Computer Science and Engineering
University of Washington
tracyk@cs.washington.edu

Abstract

We consider the problem of interleaving sequences of real numbers in order to maximize the minimum, over all prefixes p of the interleaved sequence, of the sum of the numbers in p. A simple and efficient solution is given. This problem is motivated by a resource scheduling application, and a special case of the scheduling problem is reduced to the interleaving problem.

1 Introduction

As processors get faster relative to memory speeds, file system I/O is increasingly a bottleneck to performance. Recent studies have shown that an application program's I/O overhead can be significantly reduced using integrated prefetching and caching policies [1, 2]. These techniques, augmented by mechanisms to allocate cache space among multiple processes, were shown empirically to improve the performance of multi-programmed workloads as well, assuming standard, fair scheduling mechanisms are used to arbitrate the processes' competing prefetch requests and processing demands. A natural question arises: can performance improve further if we assume that not only are prefetching and caching decisions under the control of a single manager, but that the scheduling of I/O resources (i.e. the order in which different processes' prefetch requests are served) and processing resources (the order in which the processes' computations are executed) are integrated into the policy as well?

Consider the following example. Suppose each of two processes, P and Q, start at time 0 and request data in a cyclic fashion. For simplicity, suppose each processes' data file consists of only two blocks. Thus, process P issues the request sequence $p_1, p_2, p_1, p_2, \ldots$ and Q issues the request sequence $q_1, q_2, q_1, q_2, \ldots$. Suppose further that 10 units of time are required to fetch a block of data into the cache, and that each process computes for 1 unit of time after each request for a block of data. The requested block must be present in the cache for the computation to proceed, of course.

P's and Q's data blocks will be brought into the cache alternately under a round-robin policy, say in the order p_1, q_1, p_2, q_2; these fetches complete at times 10, 20, 30, and 40, respectively. Until P's last block of data is available at time 30, only 2 units of work can be completed by the CPU. P is able to complete one unit of work (on block p_1 at time 11) and Q completes one unit of work (on block q_1 at time 21). Finally at time 30, P's entire data set is resident in the cache and it can run unhindered by I/O stalls. The same happens for Q at time 40.

If instead we favor one of the processes, say P, by devoting resources to it exclusively, we reach a state sooner in which the processor can be fully utilized. If the blocks are fetched in the order p_1, p_2, q_1, q_2, P can run without stalling on I/O starting at time 20. Q's full data set still becomes resident at time 40.

A simplified version of this integrated scheduling problem reduces to the following combinatorial problem. The reduction is outlined in section 4; full details are given in section 8. Imagine that you are given a set of several independent streams of transactions (drafts and deposits) on a checking account that is backed by a savings account. Any time the checking account is overdrawn, the overdraft must be covered from savings. Your goal is to produce an ordering of the transactions that

1. respects the orders of the individual streams, and
2. minimizes the amount that has to be transferred from savings to cover overdrafts in the checking account.

In section 7, we present a simple and efficient algorithm that finds an exact solution. The algorithm requires $O(n \log m)$ arithmetic operations, where n is the total number of transactions in all of the m sequences.

2 Motivation and background

Recent research into maximizing file system performance shows that many applications' I/O demands can be disclosed to the file system in advance [2, 3]. Suppose multiple processes share a cache backed by a storage device, and that the system has advance knowledge of the processes' sequences of requests for items residing on the backing store. *Prefetching* of items into the cache (initiating a transfer before a process faults on a missing item, and overlapping the servicing of processes' requests with the transfer) is allowed. Suppose there are no constraints regarding the interleaved servicing of the requests of the different processes, other than that the requests of an individual process must be served in order. The goal is to minimize the completion time of the last process to finish.

The general problem described above appears to be a difficult one, even in the case of a single process. In the single-process case, it can be shown [1] that any time a block must be evicted, an optimal choice is available: that block which is not needed for the longest time among all blocks present in the cache. (This rule is derived from Belady's optimal offline paging algorithm [4].) Cao *et. al* [1] also showed that it is always best to choose the first missing block when deciding which block to prefetch, and that a prefetch should never be initiated for a block that is not needed until after the next request of the block that would be evicted. These rules provide guidance in determining a prefetching and caching schedule. They uniquely determine which blocks to fetch and to evict, given that a prefetch is to be scheduled. However, they do not completely determine *when* a prefetch should be initiated. New and possibly better eviction choices arise as the request sequence is served. Cao *et. al* derived an algorithm for the single-process case with performance within a factor $\min(2, 1 + F/K)$ of optimal, where F is the time to fetch a block relative to the inter-request application compute time and K is the cache size in blocks. For typical systems, $F/K \leq .02$, so this performance is very close to optimal. No polynomial-time exact solution is known.

Kimbrel *et. al* [5] considered algorithms for the single-process case in a practical setting, simulating the algorithms' behaviors using file access sequences from real programs. One of their algorithms, dubbed Forestall, determines whether to prefetch (subject to the rules described above) based only on its estimate of the likelihood of a stall given its current cache state. The algorithm does not weigh this likelihood against the benefits of waiting to prefetch in order make a better eviction decision (and thus possibly avoid stalling at a later point in the schedule). They found that this algorithm performs at least as well as the theoretically near-optimal algorithm of Cao *et. al*. This suggests that in practice, it is not necessary to solve the difficult problem of determining exactly when each prefetch should occur based on an optimal or near-optimal sequence of eviction choices. It is sufficient to prefetch whenever a stall is imminent given the current cache contents, and to delay prefetching if the cache contains the blocks needed in the near future so that no stall is imminent.

Thus, in the more difficult multiple process case, the algorithm given here may well lead to a practical prefetch scheduler (when combined with the Forestall algorithm of Kimbrel *et. al*), even though it is designed without considering the effects of cache evictions (as will become clear in section 3). Forestall will determine whether prefetching is needed; the algorithm given here determines an order in which to prefetch the multiple processes' blocks, once the decision to prefetch has been made.

3 Formal problem statement

The general problem is formalized as follows:

- Let $B = B^1 \cup \ldots \cup B^m$ be a collection of disjoint sets of blocks residing on the backing store.
- A *reference sequence*, or *request sequence*, is an ordered sequence of references $R^k = r_1^k, r_2^k, \ldots r_{|R^k|}^k$, where each $r_i^k \in B^k$.
- There are m separate reference sequences $R^1, \ldots, R^m$.
- There is a cache of size K that contains at most K blocks in B at any time.
- Fetching a block from a disk into the cache takes F time units.

The references in each sequence R^k must be *served* in order. A single reference can be served in one unit of time. However, in order for a reference to be served, it must be in the cache. We imagine that for each reference sequence there is a *cursor* that at any time points to the next request to be served. If this request is for a block that is in the cache, the cursor can be advanced by one request during the next time unit. If several cursors point to blocks that are present in the cache, one and only one of them can be advanced in a single time unit. If all requests pointed to by the cursors are for blocks that are not in the cache, the processing *stalls* until one of the missing blocks arrives in the cache (i.e., until the fetch for that block completes). Note that, to the extent that the cursors are advancing, prefetches can overlap the serving of requests.

There are two constraints on the prefetches performed:

1. If a fetch of block b is initiated at time t and the cache contains K blocks at that time, some block b' in the cache must be *evicted* to make room for the incoming block. Neither the fetched block b nor the evicted block b' is available during the F time units between t and $t + F$ in which the fetch occurs.
2. The fetches are sequential: If a fetch is initiated for a block at time t, no other fetch can be initiated until time $t' \geq t + F$.

The goal of a multi-process prefetching and caching algorithm is to construct, on input request sequences $\{R^k\}$, a schedule for prefetching and serving requests that minimizes the elapsed time required to serve all of the R^k; this elapsed time is equal to $\sum_{k=1}^{m} |R^k|$ plus the total stall time.

The schedule specifies

- which blocks to fetch,
- when to fetch them,
- which cache blocks to evict, and

- when to service each request.

In this paper, we solve this problem for the special case of an unbounded cache; that is, we assume cache evictions are never necessary. (However, for reasons described in the previous section, we believe that this algorithm is nonetheless practical.) We will thus be concerned only with the order in which to fetch blocks into the cache, and not which blocks to evict. The algorithm presented here will work no matter what set of blocks is contained in the cache initially.

At any time during the processing of the requests, for each request sequence there is some distance from the cursor to the first *hole* (block missing from the cache) in that sequence. We refer to the requests at or following the cursor and preceding the first hole as *uncovered*. Uncovered requests can be thought of as work available to the processor; if a total of U requests are uncovered in all sequences, then the cursors can be advanced a total of U times before all cursors reach holes and a stall is incurred. Clearly, an optimal prefetching algorithm can be assumed to always fill the first hole (fetch the first missing block) in some request stream. Since we assume the cache is infinite, it never pays to wait and leave the storage device idle before initiating a prefetch; this would help only in order to make a better eviction decision if the cache were bounded. Thus we assume that a fetch is initiated at time 0, and every F time units thereafter, until the last hole is filled.

4 A reduction

As mentioned previously, to minimize the overall completion time, we can focus on minimizing the total stall time, i.e., the number of time steps during which no request is served (because all cursors are blocked by holes and no request can be served until the current fetch completes). If filling a hole in some sequence uncovers F requests (including the hole and all requests up to but not including the next hole in the sequence), the schedule "breaks even" in terms of uncovered requests, since it takes F steps to fill the hole, and F uncovered requests for blocks already present in the cache can be served concurrently. Any greater number represents a net gain of uncovered requests from the time at which the fetch is initiated until it completes; fewer than F requests uncovered will decrease the amount of work available to the processor, and increase the chance of a stall. If there are $U < F$ uncovered requests at some time iF (i.e. the i^{th} fetch has just completed), then only the U uncovered requests can be served before the next fetch completes at time $(i+1)F$ and more requests are uncovered; $F - U$ steps will be spent stalling.

We thus consider each request sequence to be simply a sequence of numbers. Corresponding to a request sequence containing u_1 uncovered requests, followed by a hole, then $u_2 - 1$ cached blocks and then another hole, etc., we have the sequence $u_1, -F, u_2, -F, \ldots$. Intuitively, it costs F steps to fill a hole; this cost must be paid before the benefit (that of being able to serve the requests

uncovered) can be reaped. The problem reduces to the following problem: We are given a list of several independent streams of transactions (drafts and deposits) on a checking account, which is backed by a savings account. Whenever the checking account is overdrawn, the deficit must be made up out of savings. We need to interleave the transactions in an order respecting the orders of the individual streams and minimizing the amount that has to be transferred from savings to cover overdrafts in the checking account. (Each time a dollar is moved from savings into checking to cover a check, one unit of stall time is incurred.) We will give a more formal description of this reduction in section 8; first, we introduce some notation and derive a solution to the new problem.

5 Reduced problem statement

Definition: Given a sequence $w = w_1 \dots w_n$ of real numbers, the *depth* $D(w)$ of w is

$$\min_{0 \le i \le n} \sum_{j=1}^{i} w_j$$

and the *net* $N(w)$ of w is

$$\sum_{i=1}^{n} w_i.$$

We denote the net $N(w_1 \dots w_i)$ of a prefix $w_1 \dots w_i$ of w by $N(w, i)$ and similarly denote the depth of a prefix; we will also speak of the "depth of w at index i" or the "net of w at index i" when the meaning is clear. Notice that in comparing two sequences, the sequence whose depth is a greater number is the *shallower* of the two, since a sequence's depth is a non-positive number.

Given a set $W = \{w^k = w_1^k \dots w_{n_k}^k : 1 \le k \le m\}$ of m sequences of numbers, an *interleaving* I of W is a sequence $I_1 \dots I_n$, where $n = \sum_{k=1}^{m} n_k$, such that there is a one-to-one map M from

$$\{(k, i) : 1 \le k \le m, 1 \le i \le n_k\}$$

to $[1..n]$ such that

1. for all $1 \le k \le m$, for all $1 \le i_1 < i_2 \le n_k$, $M(k, i_1) < M(k, i_2)$, and
2. for all $1 \le k \le m$, for all $1 \le i \le n_k$, $I_{M(k,i)} = w_i^k$.

For a sequence $w = w_1 \dots w_n$, let $B(w)$ and $R(w)$ denote the shortest non-empty prefix (if it exists) of w that sums to a non-negative value and the remaining suffix, respectively; that is, $B(w) = w_1 \dots w_l$ and $R(w) = w_{l+1} \dots w_n$, where

$$l = \min_{1 \le i \le n} \{i : N(w, i) \ge 0\}$$

if $\{i : N(w_1 \ldots w_i) \geq 0 \ \& \ \ 1 \leq i \leq n\} \neq \emptyset$.

Lemma 1 *Any suffix $w_i \ldots w_{|B(w)|}$ of $B(w)$ sums to a non-negative value.*

Proof: Since $B(w)$ is the shortest non-empty prefix of w with a non-negative net, $N(w, i-1) \leq 0$, with equality holding only in the case $i = 1$. (For any i, we take $w_i \ldots w_{i-1}$ to be the empty sequence.) □

Definition: Let $\mathcal{I}(W)$ denote the set of all interleavings of W. We seek an interleaving I such that $D(I)$ is maximum. Let $\mathcal{I}^*(W)$ denote the set of optimal interleavings; that is,

$$\mathcal{I}^*(W) = \{I \in \mathcal{I}(W) : D(I) = D^*(W)\}$$

where

$$D^*(W) = \max_{I \in \mathcal{I}(W)} D(I).$$

6 Solving the reduced problem

The algorithm to find a "shallowest" interleaving is the following: consider each of the input sequences, and choose that one (call it w) with the shallowest prefix $B(w)$. Output $B(w)$, replace w by the suffix that remains after removing $B(w)$, and repeat. The algorithm runs into trouble, however, if none of the input sequences has a prefix with a non-negative sum. In this case, a dual construction allows the processing of the remaining sequences by considering suffixes with non-positive sums.

Lemma 2 *(Shallowest first) Let W be a set of sequences, and suppose that $B(w^k)$ exists and that for each $k' \neq k$, either $B(w^{k'})$ does not exist or $D(B(w^k)) \geq D(B(w^{k'}))$. Let $l = |B(w^k)|$. Then there is some interleaving $I \in \mathcal{I}^*(W)$ such that $M(k, i) = i$ for all $1 \leq i \leq l$, where M is the map associated with I.*

Proof: We first show that for every interleaving $I \in \mathcal{I}(W)$, $D(I) \leq D(B(w^k))$. Let I be an interleaving of W, given by map M. Let k' be the index of the sequence $w^{k'}$ such that $B(w^{k'})$ "finishes first" in I, i.e. $M(k', |B(w^{k'})|) < M(k'', |B(w^{k''})|)$ for all $k'' \neq k'$ such that $B(w^{k''})$ exists. Since each $w^{k''}$, $k'' \neq k'$, contributes a non-positive sum to

$$D(I, M(k', |B(w^{k'})|) = \min_{0 \leq i \leq M(k', |B(w^{k'})|)} \sum_{j=1}^{i} I_j$$

(whether $B(w^{k''})$ exists or not), we have

$$D(I) \leq \min_{0 \leq i \leq |B(w^{k'})|} \sum_{j=1}^{i} w_j^{k'} = D(B(w^{k'})).$$

Since $D(B(w^k)) \geq D(B(w^{k'})$, the claim follows.

Next, we show that any interleaving I, given by map M, that doesn't satisfy the claim of the lemma can be transformed into an interleaving I' (given by a map M') that does, with $D(I') \geq D(I)$. I' is obtained from I by "moving up" the entries in $B(w^k)$ to the beginning of the interleaving, without changing the respective orderings among the entries of $R(w^k)$ and the sequences other than w^k. For $i \leq l$, let $M(k,i) = i$ and for $i > l$, let $M'(k,i) = M(k,i)$. For each (k',i') such that $k' \neq k$, let $M'(k',i') = M(k',i') + |\{i \leq l : M(k,i) > M(k',i')\}|$. By the preceding argument, the net value $N(I', M'(k,i)) = N(I',i)$ for each $1 \leq i \leq l$ is at least as great as the depth $D(I)$ of the original interleaving. For each $k' \neq k$, each entry $w_{i'}^{k'}$ has a (possibly empty) suffix of $B(w^k)$ moved ahead of it in I'. By lemma 1, that suffix has a non-negative net, so that $N(I'_1 \ldots I'_{M'(k',i')}) \geq N(I_1 \ldots I_{M(k',i')})$. Thus the overall depth of I' is no smaller than that of I, since at each index of I' there is an index of I with net value at least as small. □

In order to apply this lemma to the interleaving problem, it is necessary that at least one of the sequences to be interleaved has a non-empty prefix with a non-negative net. When this fails, we use a dual notion. Notice that for a sequence $w = w_1 \ldots w_i w_{i+1} \ldots w_n$, $N(w_1 \ldots w_i) = N(w) - N(w_{i+1} \ldots w_n)$. Thus maximizing the minimum prefix sum (of an interleaving) is equivalent to minimizing the maximum suffix sum. This motivates the following:

Definition: For sequence $w = w_1 \ldots w_n$, let $B'(w)$ and $R'(w)$ denote the shortest non-empty suffix (if it exists) of w that sums to a non-positive value and the remaining prefix, respectively; that is, $B'(w) = w_{l+1} \ldots w_n$ and $R'(w) = w_1 \ldots w_l$, where

$$l + 1 = \max_{1 \leq i \leq n} \{i : N(w_i \ldots w_n) \leq 0\}$$

if $\{i : N(w_i \ldots w_n) \leq 0 \ \& \ 1 \leq i \leq n\} \neq \emptyset$. The *height* $H(w)$ of any sequence $w = w_1 \ldots w_n$ is

$$\max_{0 \leq i \leq n} \sum_{j=i+1}^{n} w_j.$$

A dual argument to Lemma 2 yields the following:

Lemma 3 *(Lowest last) Let W be a set of sequences, and suppose that $B'(w^k)$ exists and that for each $k' \neq k$, either $B'(w^{k'})$ does not exist or $H(B'(w^k)) \leq H(B'(w^{k'}))$. Let $l = |R'(w^k)|$. Then there is some interleaving $I \in \mathcal{I}^*(W)$ such that $M(k,i) = (\sum_{j=1}^{m} n_j) - n_k + i$ for all $l+1 \leq i \leq n_k$, where M is the map associated with I.*

7 The algorithm

Notice that, since every non-empty w is both a non-empty prefix and a non-empty suffix of itself, for every non-empty w either $B(w)$ or $B'(w)$ exists. Notice also that if $B(w)$ does not exist,

then removing the suffix $B'(w)$ from w will not change this; i.e. $B(R'(w))$ does not exist. These observations, along with Lemmas 2 and 3, imply that an optimal interleaving of W is obtained by the following algorithm:

Repeat until for each k, $1 \leq k \leq m$, either
w^k is empty or $B(w^k)$ does not exist:
  Let k satisfy $D(B(w^k)) \geq D(B(w^{k'}))$
  for all k' such that $B(w^{k'})$ exists.
  Output $B(w^k)$ and replace w^k with $R(w^k)$.
Initialize a stack S to the empty stack.
Repeat until for each k, $1 \leq k \leq m$, w^k is
empty:
  Let k satisfy $H(B'(w^k)) \leq H(B'(w^{k'}))$
  for all k' such that $w^{k'}$ is not empty.
  Push $B'(w^k)$ on S (in reverse order) and
  replace w^k with $R'(w^k)$.
While S is not empty output $pop(S)$.

A straightforward modification of the algorithm outputs the map M by which the interleaving is obtained from the input set W. The algorithm can be implemented to use $O(n \log m)$ operations (comparisons, additions, and assignments), where n is the sum of the lengths of the input sequences (and equal to the length of the output sequence), and m is the number of input sequences. This is achieved even in the case that each $B(w^k)$ and $B'(w^k)$ is short (length bounded by a constant). A linear scan of each $B(w^k)$ can determine its length and depth (if it exists; if not, a linear scan of all of w^k determines this, and w^k need not be considered again until the second loop is entered.) These records can be stored in a priority queue keyed on the depth. On each iteration of the first loop, a delete maximum operation determines which $B(w^k)$ to output, and which w^k to examine in order to insert (a description of) the new $B(w^k)$ into the queue. The second loop can be handled similarly. Producing the output has a total cost of $O(n)$. Thus, the most expensive operations are the $O(n)$ insert and delete maximum (or minimum) operations on the priority queue, each with a cost of $O(\log m)$ (see, for example, [6]).

Thus we have the following.

Theorem 4 *The above algorithm finds an optimal interleaving of the m sequences $W = \{w^k = w^k_1 \ldots w^k_{n_k} : 1 \leq k \leq m\}$ in time $O(n \log m)$ in the unit cost RAM model.*

8 Formalizing the reduction

We return now to the reduction of the prefetching and scheduling problem to the checking and savings account problem. Lemma 2 allows us to assume that the initial non-negative entries of all

m sequences (i.e. the numbers of initially uncovered requests) occur first in the interleaving (in an arbitrary order; say in the same order as that in which the input sequences occur). Lemma 2 also allows us to assume that each subsequent pair $(-F, u_i^j)$ corresponding to the i^{th} hole in the j^{th} request sequence occurs consecutively in an interleaving I, since the single positive value u_i^j is a zero-depth prefix. It is thus easy to determine a prefetching schedule that corresponds naturally to an interleaving I with associated map M, such that I_{m+2i} is the number of requests uncovered by the i^{th} fetch; for each prefetching schedule there is a unique such corresponding map specifying an interleaving. In the following, we assume that an interleaving I is known and has been used to determine a prefetching schedule. We claim that $|D(I)|$ is the total stall time of the prefetching schedule.

One direction (the lower bound on total stall time) is easy: since a total of $N(I, m+2i) + iF$ requests are uncovered (initially and by prefetch operations) before time $(i+1)F$, at most $N(I, m+2i) + iF$ requests can be served by time $(i+1)F$. Thus, the stall time accumulated up to time $(i+1)F$ is at least $\max(0, (i+1)F - (iF + N(I, m+2i))) = \max(0, F - N(I, m+2i))$. Unless $m+2i = |I|$, we have that $I_{m+2i+1} = -F$, so that the stall time is at least $\max(0, -N(I, m+2i+1))$. Taking the minimum value over i of $N(I, m+2i+1)$ (and noting that the minimum net is achieved at such an index since I_{m+2i} is positive for each i) yields the lower bound.

We now show that this bound on the stall time can be met. We show by induction on i that at time iF,

1. the number of requests left uncovered and available for servicing between times iF and $(i+1)F$ is $N(I, m+2i) - D(I, m+2i)$, and
2. the accumulated stall time is $|D(I, m+2i)|$.

The basis ($i = 0$) is trivial. For the induction, assume the hypothesis is true for i.

Case 1: $D(I, m+2i+2) = D(I, m+2i)$. In this case, $N(I, m+2i+1) \geq D(I, m+2i)$ so that $N(I, m+2i) - D(I, m+2i) \geq F$, since $I_{m+2i+1} = -F$. Thus there are at least F requests uncovered at time iF, and no further stalling is incurred between times iF and $(i+1)F$. F requests are served between times iF and $(i+1)F$, the accumulated stall time is (unchanged) $|D(I, m+2i)| = |D(I, m+2i+2)|$, and the number of requests left uncovered is $N(I, m+2i) - D(I, m+2i) - F + I_{m+2i+2} = N(I, m+2i+2) - D(I, m+2i+2)$ as needed.

Case 2: $D(I, m+2i+2) < D(I, m+2i)$. In this case, $N(I, m+2i) - D(I, m+2i) < F$, and the number of additional stall steps incurred is $F - (N(I, m+2i) - D(I, m+2i)) = D(I, m+2i) - N(I, m+2i+1) = D(I, m+2i) - D(I, m+2i+2)$ so that the total is $|D(I, m+2i+2)|$. The number of requests left uncovered at time $(i+1)F$ is $I_{m+2i+2} = N(I, m+2i+2) - N(I, m+2i+1) = N(I, m+2i+2) - D(I, m+2i+2)$ as needed.

9 Conclusion

In this paper, we have considered an abstract combinatorial problem, "sequence interleaving," derived from a multiple resource scheduling problem. A simple and efficient algorithm was given for the sequence interleaving problem. The sequence interleaving problem corresponds to a simplification of the integrated scheduling problem, which appears to be difficult. However, there is reason to believe that a solution to the simplified problem will perform well in practice, as discussed in section 2. Future work includes testing this hypothesis by comparing the algorithm to existing approaches, using simulation based on traces of real programs' resource demands.

References

[1] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of Integrated Prefetching and Caching Strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995, pp. 188–197.

[2] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995, pp. 79–95.

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

[4] L.A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna Karlin, and Kai Li. A Trace-driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, October 1996, pp. 19–34.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison–Wesley, Reading, MA, 1983, pp. 135–145.