Integrated Prefetching and Caching in Single and Parallel Disk Systems

Susanne Albers *
Institute for Computer Science
Freiburg University
Georges-Köhler-Allee 79
79110 Freiburg, Germany
salbers@informatik.uni-freiburg.de

Markus Büttner
Institute for Computer Science
Freiburg University
Georges-Köhler-Allee 79
79110 Freiburg, Germany
buettner@informatik.uni-freiburg.de

ABSTRACT

We study integrated prefetching and caching in single and parallel disk systems. There exist two very popular approximation algorithms called *Aggressive* and *Conservative* for minimizing the total elapsed time in the single disk problem. For D parallel disks, approximation algorithms are known for both the elapsed time and stall time performance measures. In particular, there exists a D-approximation algorithm for the stall time measure that uses $D-1$ additional memory locations in cache.

In the first part of the paper we investigate approximation algorithms for the single disk problem. We give a refined analysis of the *Aggressive* algorithm, showing that the original analysis was too pessimistic. We prove that our new bound is tight. Additionally we present a new family of prefetching and caching strategies and give algorithms that perform better than *Aggressive* and *Conservative*.

In the second part of the paper we investigate the problem of minimizing stall time in parallel disk systems. We present a polynomial time algorithm for computing a prefetching/caching schedule whose stall time is bounded by that of an optimal solution. The schedule uses at most $3(D-1)$ extra memory locations in cache. This is the first polynomial time algorithm for computing schedules with a minimum stall time. Our algorithm is based on the linear programming approach of [1]. However, in order to achieve minimum stall times, we introduce the new concept of synchronized schedules in which fetches on the D disks are performed completely in parallel.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and problems

General Terms

Algorithms

Keywords

Magnetic disk systems, prefetching, caching

1. INTRODUCTION

In today's computer systems there is a growing gap between processor speed and memory access time. Therefore an effective utilization of caches is increasingly important. Prefetching and caching are well-known and extensively studied techniques to improve the performance of memory hierarchies. In prefetching missing memory blocks are loaded from slow memory, e.g. a disk, into cache before their actual reference. Caching strategies try to keep actively referenced blocks in cache. The goal of both tools is to reduce processor stall times that are incurred when requested data is not available in cache. Most of the previous work on prefetching and caching investigated these two techniques separately, see e.g. [3, 4, 7, 8, 15, 18, 19] for some selected papers, although there is a strong correlation. Prefetching blocks too early can cause the eviction of blocks from cache referenced in the near future. Caching blocks too long can diminish the effect of prefetching.

In recent years, initiated by a paper of Cao et al. [5], there have been a number of studies that integrate prefetching and caching. The goal is to design strategies that coordinate prefetching and caching decisions. Both theoretical and experimental studies were presented [5, 6, 9, 10, 11, 12, 13, 14, 16, 17]. It was demonstrated that an integration of prefetching and caching leads to a substantial improvement in systems performance.

Cao et al. [5] introduced a model for integrated prefetching and caching that we will also use in this paper. We are given a request sequence $\sigma = r_1, \ldots, r_n$ consisting of n requests. Each request specifies a block in the memory system. We first assume that all blocks reside on a single disk. To serve a request the requested block must be in cache. The cache can simultaneously store k blocks. Serving a request to a block in cache takes 1 time unit. If a requested block is not in cache, then it must be fetched from disk, which takes F time units. A fetch operation may overlap with the service of

*Work supported by the Deutsche Forschungsgemeinschaft, project project AL 464/3-1, and by the EU, projects APPOL and APPOL II.

SPAA'03, June 7–9, 2003, San Diego, California, USA.

requests to blocks already in cache. If a fetch, i.e. a prefetch, of a block is initiated at least F requests before the reference to the block, then the block is in cache at the time of the request and no processor stall time is incurred. If the fetch is started only $i, i < F$, requests before the reference, then the processor has to stall for $F - i$ time units until the fetch is finished. When a fetch operation is initiated, a block must be evicted from cache to make room for the incoming block. Thus a prefetch operation critically affects the cache configuration in that we must also drop a block. The goal is to minimize the total *processor stall time* incurred on the entire request sequence. This is equivalent to minimizing the *elapsed time*, which is the sum of the processor stall time and the length of the request sequence. We point out here that the input σ is completely known in advance.

To illustrate the problem, consider a small example. Let $\sigma = b_1, b_2, b_3, b_4, b_4, b_5, b_1, b_4, b_4, b_2$. Assume that we have a cache of size $k = 4$ and that initially blocks b_1, b_2, b_3 and b_4 reside in cache. Let $F = 4$. The first missing block is b_5. We could initiate the fetch for b_5 when starting the service of the request to b_2. The fetch would be executed while serving requests b_2, b_3, b_4 and b_4 and completed in time. However, when starting this fetch, we can only evict b_1, which is requested again after b_5. To load b_1 we incur 3 units of stall time as the fetch can only overlap with the request to b_5. A better option is to start the fetch for b_5 at the request to b_3. We generate 1 unit of processor stall before the request to b_5 but can evict b_2, which is requested again only at the end of σ and can be fetched back without incurring any stall time. The stall time of this second solution is 1 time unit and the elapsed time is 11 time units.

Integrated prefetching and caching is equally interesting in parallel disk systems. Suppose that we have D disks and that each memory block resides on exactly one of the disks. Blocks from different disks may be fetched in parallel. When starting a fetch, we can evict any block from cache, which corresponds to the case that blocks are read-only and do not have to be written back to disk. Of course we can take advantage of the parallelism given by a multiple disk system. If the processor incurs stall time to wait for the completion of a fetch, then other fetches executed in parallel also make progress towards completion during that time. Again we wish to minimize the total stall time or elapsed time.

As an example consider two disks, where b_1, b_2, b_3 and b_4 reside on disk 1 and c_1, c_2 and c_3 reside on disk 2. Again $k = 4$ and $F = 4$. Suppose that initially b_1, b_2, c_1 and c_2 are in cache and that $\sigma = b_1, b_2, c_1, c_2, b_3, c_3, b_4$. Disk 1 initiates a fetch for b_3 at the request to b_2; it evicts b_1. Disk 2 starts a fetch for c_3 one request later and evicts b_2. Disk 1 starts a second fetch at the request to b_3 in order to load b_4. There is 1 unit of stall time before the request to b_3. The fetch on disk 2 benefits from this time unit so that no additional stall time is generated before the request to c_3. The second fetch on disk 1 incurs 2 units of stall time. The total stall time of this solution is equal to 3 time units.

Previous work: Cao et al. [5] introduced two popular algorithms, called *Conservative* and *Aggressive*, for integrated prefetching and caching in the single disk problem. *Conservative* performs exactly the same block replacements as the optimum offline paging algorithm MIN [3], while initiating a fetch at the earliest point in time that is consistent with the choice of blocks to be evicted. Cao et al. showed that *Conservative* achieves an approximation ratio of 2 with respect to the elapsed time performance measure, i.e. the elapsed time of *Conservative*'s schedule is at most twice the elapsed time of an optimal schedule. This bound is tight. The *Aggressive* algorithm starts prefetch operations as soon as possible. Whenever the algorithm is not in the middle of a fetch, it initiates a new fetch provided it can evict a block from cache that is not requested before the block to be fetched. Cao et al. proved that the approximation ratio, with respect to elapsed time, is at most $\min\{1 + F/k, 2\}$ and that this ratio is tight for $F \geq k$. Kimbrel and Karlin [12] analyzed *Aggressive* and *Conservative* in parallel disk systems and showed that the approximation ratios are essentially equal to D. They also proposed an algorithm *Reverse Aggressive*, which is the *Aggressive* algorithm on the reverse sequence, and proved that the approximation guarantee is bounded by $1 + DF/k$. Again the approximation ratios are with respect to the elapsed time measure. Extensive experimental studies, in particular on the performance of the *Aggressive* algorithm, were presented in [5, 6, 13, 14].

It was shown in [1] that optimal prefetching/caching schedules for a single disk can be computed in polynomial time. The idea is to formulate the prefetching and caching problem as a linear program and to prove that there exists an optimal solution that is integral. The approach was extended to parallel disk systems and gave a D-approximation algorithm for the stall time performance measure if the algorithm may use $D - 1$ extra memory locations in cache. Note that approximating stall time is harder than approximating elapsed time because in the stall time measure the length of the request sequence is not part of the objective function. In [2] it was shown that the linear program of [1] can be translated directly into a multicommodity flow problem.

Our contribution: We investigate approximation algorithms for single disk systems as well as algorithms for computing optimal schedules in parallel disk systems. In Section 2 we study the single disk problem and first present a refined analysis of the *Aggressive* algorithm, showing that the analysis by Cao et al. [5] was too pessimistic. We prove that *Aggressive* achieves an approximation ratio of $\min\{1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1), 2\}$ in the elapsed time measure. Compared to the bound of Cao et al. there is an additional $\lfloor \frac{k}{F} \rfloor - 1$ in the denominator of the first term. If k/F is large, which is true in most practical applications, the new bound is much lower. Kimbrel and Karlin [12] mentioned that in practice k/F is typically at least 200. We also show that our analysis is tight. For any F and k, the approximation ratio of *Aggressive* is in general not smaller than $\min\{1 + F/(k + \frac{k-1}{F-1}), 2\}$. Since *Aggressive* is the most popular algorithm for integrated prefetching and caching, it is important to know its true approximation guarantee.

We also give improved approximation ratios if k/F is small. More generally, we present a new family of algorithms for integrated prefetching and caching. The algorithms, called *Delay(d)*, delay the next fetch operation for d time units, for any fixed non-negative integer d. Setting $d = 0$, *Delay(d)* is equal to the standard *Aggressive* strategy; for $d = |\sigma|$ we obtain *Conservative*. Hence our family of algorithms bridges the gap between the two classical algorithms for prefetching and caching. As mentioned above, fetching blocks too early can have a negative influence on the cache configuration and reduce the effective size of the cache. Thus it is natural to investigate the effect of delaying fetches by some time units. We analyze *Delay(d)* for any

d and show that, surprisingly, the best choice of d gives an approximation ratio of $\sqrt{3} \approx 1.73$. Combining this strategy with *Aggressive*, we obtain an algorithm that achieves an approximation ratio of $\min\{1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1), \sqrt{3}\}$ and hence performs better than both *Conservative* and *Aggressive*.

In Section 3 we investigate the problem of minimizing stall time on D parallel disks. We present a polynomial time algorithm that, given a request sequence σ, computes a schedule whose stall time is bounded by that of an optimal solution for σ. The solution uses at most $3(D-1)$ extra memory locations in cache. In practice D is small, typically 4 or 5. Thus at the expense of slightly increasing the extra memory resources, we are able to improve the best previous approximation guarantee from D to 1. In fact our algorithm is the first polynomial time strategy for computing schedules with a minimum stall time. Our algorithm is based on the linear programming approach of [1]. However, in order to obtain solutions with a smaller stall time, we introduce the new concept of synchronized schedules in which fetches on the D disks are performed completely in parallel. We show that there exist synchronized schedules that achieve a minimum stall time provided that they may use $D-1$ extra memory locations in cache. Using linear programming we then compute an optimal synchronized solution that uses $D-1$ extra cache locations. Applying techniques from [1] we transform an optimal fractional solution into an integral solution. Compared to [1], our transformation algorithm must use a different scheme for assigning blocks to be evicted in the integral solution.

2. APPROXIMATION ALGORITHMS FOR A SINGLE DISK

Throughout this section approximation ratios refer to the elapsed time performance measure. The *Aggressive* algorithm works as follow. Whenever the algorithm is not prefetching a block, it initiates a prefetch for the next missing block in the sequence provided it can evict a block from cache that is not requested before the block to be fetched. In this case it evicts the block whose next reference is furthest in the future.

THEOREM 1. *The approximation ratio of Aggressive is at most* $\min\{1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1), 2\}$.

Our upper bound proofs in this section are based on the *dominance* concept introduced by Cao et al. [5]. Given a request sequence $\sigma = r_1, \ldots, r_n$ consisting of n requests and a prefetching algorithm A, let $c_A(A)$ be the index of the next request at time t when A serves σ. Let $H_A(i)$ be the set of blocks not present in A's cache when the next reference is r_i. Let $h_A(i,j)$, $h_A(i,j) \geq i$, be the smallest index such that exactly j different blocks in $H_A(i)$ are referenced in the subsequence consisting of request i up to (and including) request $h_A(i,j)$. Intuitively, $h_A(i,j)$ is the index of the first reference to the j-th block not present in cache after r_{i-1}. Index $h_A(i,j)$ is also referred to as A's j-th *hole*. The parameter j varies between 1 and $n-k$. Given two prefetching algorithms A and B, *A's cursor at time t dominates B's cursor at time t'* if $c_A(t) \geq c_B(t')$. We say that *A's holes at time t dominate B's holes at time t'* if $h_A(c_A(t), j) \geq h_B(c_B(t'), j)$ for all j. Combining these two definitions we say that *A's state at time t dominates B's state at time t'* if A's cursor at time t dominated B's cursor at time t' and A's holes at time t dominate B's holes at time t'. Cao et al. [5] proved the following domination lemma.

LEMMA 1. *Suppose that algorithm A (resp. algorithm B) initiates a prefetch at time t (resp. t') and both algorithms prefetch the next missing block and replace the block whose next reference is furthest in the future. Suppose that A's state at time t dominates B's state at time t'. Then A's state at time $t+F$ dominates B's state at time $t'+F$.*

PROOF OF THEOREM 1. We assume $F \leq k$ and prove an upper bound of $1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1)$ on *Aggressive*'s approximation ratio. If $F > k$, then our bound implies a 2-approximation, which was already shown by Cao et al. [5]. The global structure of our proof is similar to that of Cao et al. and we describe the difference. Let OPT be an optimal prefetching algorithm. We partition the given request sequence into phases such that each phase consists of exactly $k + \lfloor \frac{k}{F} \rfloor - 1$ consecutive requests. We prove by induction on the number of phases that the following invariant holds. During each phase i there is a time t such that *Aggressive*'s state at time t dominates OPT's state at time $t' \geq t - iF$. This implies that *Aggressive* needs at most $F \cdot$ (number of phases) more time units than OPT to serve the entire request sequence, and on the average it spends at most F extra time units in each phase. Cao et al. divided the request sequence into different phases which consisted potentially of only k requests. This resulted in a higher upper bound.

To establish the invariant consider a phase i and assume that at time t during phase i *Aggressive*'s state dominates OPT's state at time $t' \geq t - iF$. We show that for all times $\tau > 0$ such that *Aggressive* is in phase i at time $t+\tau$, $c_A(t+\tau) \geq c_{OPT}(t'+\tau)$, where $c_A(t)$ is *Aggressive*'s cursor position at time t. We distinguish two cases. (1) During phase i, *Aggressive* never evicts a block from cache that is requested again in phase i. (2) During phase i, *Aggressive* does evict blocks that are requested again in the phase. The first case is easy to analyze. While *Aggressive* serves a subsequence of requests without fetching blocks, the cursor advances one request in each time step and hence OPT's cursor cannot pass *Aggressive*'s cursor. During the service and in particular at the end of this subsequence *Aggressive*'s holes dominate OPT's holes because the k blocks in *Aggressive*'s cache are all requested before any block not in cache and thus the holes occur at the latest possible positions. While *Aggressive* performs a series of fetches, *Aggressive*'s holes always dominate OPT's holes because no further holes are introduced in the phase. Hence *Aggressive*'s cursor cannot fall behind OPT's cursor and repeating these two arguments we obtain the desired inequality.

We next consider case (2). Let $l, 1 \leq l \leq k + \lfloor \frac{F}{k} \rfloor - 1$, be the smallest index such that *Aggressive*, while processing the phase, evicts the block referenced by the l-th request in the phase. Let $t'', t'' \geq t$, be the time when *Aggressive* initiates the prefetch in which this block is evicted. Obviously $l > k$ because when *Aggressive* fetches a block during the first k requests it can always evict a block from cache that is not referenced during the next k requests. Let $l = k + j$, for some $j \geq 1$. When *Aggressive* initiates the prefetch at time t'', there can be at most j blocks missing in cache that are requested until the l-th request in the phase. If there were more than j such blocks, then *Aggressive* could evict a block

whose next request has an index larger than $k+j$. Thus the algorithm has to execute at most $j + \lfloor \frac{k}{F} \rfloor - 1 - j + 1 = \lfloor \frac{k}{F} \rfloor$ fetches (including the one just initiated) to bring all blocks into cache that are still requested in the phase. We distinguish again two cases depending on whether or not these fetches are executed immediately one after the other.

Between time t and t'' *Aggressive* never fetches a block that is not requested during the first $l-1$ requests of the phase. Otherwise ther would be a time where all blocks requested up to the l-th request are in cache and *Aggressive* initiates a fetch for a block that is requested after the block evicted. Thus all blocks fetched between t and t'' were not in cache at time t and using the same arguments as in case (1) we obtain that $c_A(t+\tau) \geq c_{OPT}(t'+\tau)$ for all τ with $\tau \leq t''-t$. At time t'' the l-th request of the phase is at least k requests away from the current request because when initiating a fetch *Aggressive* can always evict a block from cache that is not requested during the next k references. Now suppose that the at most $\lfloor \frac{k}{F} \rfloor$ fetches after time t'' are executed immediately one after the other. *Aggressive* first fetches the j', $j' \leq j$, blocks that are missing up to the l-th request of the phase. By the choice of l all these blocks were also missing at the beginning of the phase and again *Aggressive*'s cursor always dominates OPT's cursor during these fetches. After the service of at most $j'F$ additional requests after time t'' these fetches are complete. During the next $k - j'F \geq \lfloor \frac{k}{F} \rfloor F - j'F = (\lfloor \frac{k}{F} \rfloor - j')F$ requests *Aggressive* can complete all the remaining fetches for blocks in the phase and hence completes these fetches before the l-th request. No stall time is incurred and *Aggressive*'s cursor dominates OPT's cursor. During the rest of the phase *Aggressive* does not incur stall time either and again its cursor dominates OPT's cursor.

We finally consider the case that the at most $\lfloor \frac{k}{F} \rfloor$ fetches after t'' are not executed immediately one after the other. Let l' be the index of the last request in the phase such that at least one of the fetches still has to be executed but no fetch is performed during the service of the request. Let s be the time when *Aggressive* reaches this request. As in the previous paragraph we can show that *Aggressive*'s cursor dominates OPT's cursor between t'' and s: *Aggressive* first fetches missing blocks that are referenced before the l-th request of the phase and hence were missing at the beginning of the phase. *Aggressive*'s cursor cannot fall behind OPT's cursor. Then *Aggressive* fetches some blocks that are requested during the l-th request of the phase or later. This can be done without incurring any stall time because $l' \leq k < l$. To see the first inequality, observe that at time s the next k requests are in cache because no fetch is performed; however blocks requested in the phase are still missing in cache and $\lfloor \frac{k}{F} \rfloor - 1 < k$. Again *Aggressive*'s cursor cannot fall behind OPT's cursor. After time s *Aggressive* can fetch the at most $\lfloor \frac{k}{F} \rfloor$ missing blocks of the phase without generating any stall time. This is because $l > k$ implies $\lfloor \frac{k}{F} \rfloor - 1 > 0$ and hence $F \leq k-1$. *Aggressive*'s cursor dominates OPT's cursor for the rest of the phase. We conclude, as desired, $c_A(t+\tau) \geq c_{OPT}(t+\tau)$, for all τ such that *Aggressive* is still in the phase at time $t+\tau$.

The rest of the proof is identical to that of Cao et al. Let $t+t_0, t+t_1, \ldots, t+t_r$ be the cursor positions where *Aggressive* initiates fetches after time t but still within the current phase. If OPT is executing a fetch at time $t+t_q, 0 \leq q \leq r$, then let $t+t'_q$ be the time when this fetch was initiated; otherwise let $t'_q = t_q$. Cao et al. proved inductively that *Aggressive*'s state at time $t+t_q$ dominates OPT's state at time $t'+t'_q$ and that *Aggressive*'s state at time $t+t_r+F$ also dominates OPT's state at time $t'+t'_r+F$. The proof makes no assumptions on the phase length and only relies on the fact that $c_A(t+\tau) \geq c_{OPT}(t'+\tau)$, for all τ such that *Aggressive* is still in phase i. Thus we also have that *Aggressive*'s state at time $t+t_r+F$ dominates OPT's state at time $t'+t'_r+F \geq t'+t_r$. We conclude that *Aggressive* is in phase $i+1$ at time $T = t+t_r+F$ and that its state dominates OPT's state at time $t'+t_r \geq t-iF+t_r \geq T-(i+1)F$. □

THEOREM 2. *The approximation ratio of Aggressive is in general not smaller than* $\min\{1+F/(k+\frac{k-1}{F-1}), 2\}$, *for any* $F > 1$.

PROOF. We assume $F \leq k$. For $F > k$, a lower bound of 2 was already shown by Cao et al. [5]. Consider any pair F and k such that $F-1$ divides $k-1$ and let $l = \frac{k-1}{F-1}$. We construct a request sequence in phases, each consisting of $k+l$ requests. In each phase we request blocks $a_1, \ldots, a_{k-l}$. In phase $i, i \geq 1$, we request l new blocks $b_1^i, \ldots, b_l^i$ which have not been referenced before in the sequence. These are requested at the end of the phase. After the requests to a_1 we request the new blocks $b_1^{i-1}, \ldots, b_l^{i-1}$ from the previous phase, and these blocks will not be requested again during the rest of the sequence. Suppose that *Aggressive* has initially blocks $a_1, \ldots, a_{k-l}$ and $b_1^0, \ldots, b_l^0$ in its cache. Then the first three phases are as follows.

$$\begin{aligned} \sigma \;=\; & a_1, b_1^0, \ldots, b_l^0, a_2, \ldots, a_{k-l}, b_1^1, \ldots, b_l^1, && \text{//phase 1} \\ & a_1, b_1^1, \ldots, b_l^1, a_2, \ldots, a_{k-1}, b_1^2, \ldots, b_l^2, && \text{//phase 2} \\ & a_1, b_1^2, \ldots, b_l^2, a_2, \ldots, a_{k-l}, b_1^3, \ldots, b_l^3, \ldots && \text{//phase 3} \end{aligned}$$

In the first phase *Aggressive* starts fetching the missing blocks $b_1^1, \ldots, b_l^1$ after the service of a_1. It first evicts a_1 and then blocks $b_1^0, \ldots, b_{l-1}^0$ since the latter are not requested again. *Aggressive* needs $l \cdot F = \frac{k-1}{F-1} \cdot F = k-1+l$ time units to complete the fetches and hence has one unit of stall time before the service of b_l^1. *Aggressive* then loads the missing block a_1 by evicting b_l^0 and incurs $F-1$ units of stall time. At the beginning of phase 2 *Aggressive* has blocks $a_1, \ldots a_{k-l}$ and $b_1^1, \ldots, b_l^1$ in its cache. The situation is the same as at the beginning of phase 1 except that the b_j^1 take the role of the b_j^0 and the b_j^2 take the role of the $b_j^1, j = 1, \ldots, l$. The same pattern repeats during the other phases. Thus *Aggressive* needs $k+l+F$ time units to serve a phase. On the other hand, an optimal strategy starts fetching the missing blocks in any phase i after the service of b_1^{i-1} and can thus evict the blocks $b_1^{i-1}, \ldots, b_l^{i-1}$ to load $b_1^i, \ldots, b_l^i$. OPT incurs two units of stall time in each phase and needs $k+l+2$ time units for any phase. The ratio of *Aggressive*'s time to the optimal time is $1 + (F-2)/(k+\frac{k-1}{F-1}+2)$ and this can be arbitrarily close to the stated bound. □

In addition to the *Aggressive* algorithm Cao et al. [5] proposed the *Conservative* strategy. *Conservative* performs exactly the same replacements as the optimum offline paging algorithm MIN [3] while initiating a fetch at the earliest opportunity that is consistent with the choice of the block to be evicted. We now present a family of algorithms that contains *Aggressive* and *Conservative* at two ends of its spectrum. Using this family we construct an algorithm that performs

better than *Aggressive* and *Conservative*. Let d be a non-negative integer. Intuitively the following algorithm delays a fetch for d time units.

Algorithm Delay(d): Let r_i be the next request to be served and $r_j, j \geq i$, the next reference where the requested block is missing in cache. If all blocks in cache are requested before r_j, serve r_i without initiating a fetch. Otherwise let $d' = \min\{d, j-i\}$ and let b be the block whose next request is furthest in the future after request $r_{i+d'-1}$. Initiate a fetch for r_j at the earliest point in time after r_{i-1} such the evicted block b is not requested again before r_j.

Obviously, for $d = 0$ we obtain the standard *Aggressive* strategy. For $d = n, n$ being the length of the request sequence, we obtain the *Conservative* algorithm. Before proving the next theorem, we mention a few implications.

THEOREM 3. *For any non-negative integer d, Delay(d) achieves an approximation ratio of $c = \max\{\frac{d+F}{F}, \frac{d+2F}{d+F}, \frac{3(d+F)}{d+2F}\}$.*

COROLLARY 1. *Setting $d_0 = \lfloor \frac{1}{2}(\sqrt{3}-1)F \rfloor$, the approximation ratio c_0 of* Delay(d_0) *tends to $\sqrt{3}$.*

Algorithm Combination: If $c_0 < 1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1)$, execute *Delay($d_0$)*, otherwise execute the standard *Aggressive* strategy.

COROLLARY 2. *The approximation ratio of Combination is $\min\{1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1), c_0\}$, which tends to $\min\{1 + F/(k + \lfloor \frac{k}{F} \rfloor - 1), \sqrt{3}\}$.*

PROOF OF THEOREM 3. In the following we call our approximation algorithm *DL* for short, omitting the given parameter d. We partition the prefetching/caching schedule by *DL* and OPT into segments S^i_{DL} and $S^i_{OPT}, i \geq 1$, such that *DL*'s state at the end of S^i_{DL} dominates OPT's state at the end of S^i_{OPT} and the length of S^i_{DL} is at most c times the length of S^i_{OPT}, where $c = \max\{\frac{d+F}{F}, \frac{d+2F}{d+F}, \frac{3(d+F)}{d+2F}\}$. This establishes the theorem. The segments S_{DL} have the property that *DL* is never in the middle of a fetch at the end of S^i_{DL}. Suppose that $S^1_{DL}, \ldots, S^i_{DL}$ and $S^1_{OPT}, \ldots, S^i_{OPT}$ have been constructed so far. Let t be the time at the end of S^i_{DL} and t' be the time at the end of S^i_{OPT}. We show how to construct the next segments S^{i+1}_{DL} and S^{i+1}_{OPT}. If we are at the beginning of the request sequence and no segments have been constructed so far, we set $t = t' = 0$ and show how to build up the first segments.

DL's next segment starts immediately after t and OPT's next segment starts immediately after t'. We have to determine where the segments end and use s to denote the end of *DL*'s segment and s' to identify the end of OPT's segment. If at time t all k blocks in *DL*'s cache are requested before the next missing block, the segments are easily specified. Suppose that *DL* serves δ requests after t without initiating a fetch because all blocks in cache are requested before the next missing block. Then *DL*'s cursor at time $s = t+\delta$ dominates OPT's cursor at time $s' = t' + \delta$ and *DL*'s holes at time s also dominate OPT's holes at time s' because *DL*'s holes occur at the latest possible positions. We have the desired domination and the two segments have in fact the same length.

In the following we always assume that at time t there is a block in *DL*'s cache that is referenced again only after the next block to be fetched and hence *DL* can initiate a fetch. Assume that *DL* needs $D_1, D_1 \leq d + F$, time units after t to complete the next fetch. If OPT does not initiate a fetch during the next D_1 time units after t', then we are done. *DL*'s cursor at time $s = t + D_1$ dominates OPT's cursor at time $s' = t' + D_1$. This is obvious if *DL* does not incur stall time to complete the fetch. If *DL* does incur stall time, then *DL* fetches the block referenced right after $t+D_1$. OPT's cursor cannot pass *DL*'s cursor because *DL*'s holes at time t dominate OPT's holes at time t'. Since OPT's holes do not change between t' and s' *DL*'s holes at time s also dominate OPT's holes at time s'. Again we have the desired domination and *DL*'s and OPT's segments have the same length.

We therefore assume in the following that OPT initiates a fetch during the next D_1 time units after t'. Suppose that *DL* serves exactly d_1 requests after t and that OPT serves d'_1 after t' before initiating the next fetch. If $d'_1 \leq d_1$, the analysis is simple. *DL*'s state at time $t+d_1$ dominates OPT's state at time $t' + d'_1$ and by the Lemma 1 *DL*'s state at time $s = t + d_1 + F = t + D_1$ dominates OPT's state at time $s' = t' + d'_1 + F$. The ratio of *DL*'s segment length to OPT's segment length is at most $D_1/(d'_1 + F) \leq (d + F)/F$. If $d'_1 > d_1$ but $d'_1 \leq d$, then let r_i be the next request to be served by *DL* and r_j be the location of the next hole at time t. Set $\bar{d} = \min\{j - i, d'_1\}$. Imagine we would modify *DL* as follows. After time t *DL* serves $\bar{d}$ requests before initiating a fetch for r_j. During this fetch it evicts the block whose next reference is furthest in the future. Since *DL*'s state at time t dominates OPT's state at time t', the modified algorithm's state at time $t+\bar{d}$ dominates OPT's state at time $t+d'_1$. By Lemma 1 the modified algorithm's state at time $t + \bar{d} + F$ dominates OPT's state at time $t' + d'_1 + F$. By definition the original *DL* algorithm may delay a fetch for d requests and hence the block evicted during the first fetch after t is equal to the block evicted by the modified algorithm during the first fetch after t. We obtain that *DL*'s holes at time $s = t + D_1$ dominate OPT's holes at time $t' + d'_1 + F$, which are equal to OPT's holes at time $s' = \min\{t' + D_1, t' + d'_1 + F\}$. Also, *DL*'s cursor at time s dominates OPT's cursor at time s' because if *DL* incurs stall time to complete the fetch then OPT's cursor cannot pass because its holes were dominated by *DL*'s holes. In summary we have domination and the ratio of the segment length is upper bounded by $D_1/F \leq (d + F)/F$.

In the remainder of this proof we assume $d'_1 > d$. If at time $t + D_1$ the k blocks in *DL*'s cache are all referenced before the next missing block, then the segments are easily determined. OPT needs $D'_1 = d'_1 + F$ time units to complete the first fetch after t'. If *DL* does not incur stall time to complete the first fetch, then its cursor at time $s = t + D_1$ dominates OPT's cursor at time $t' + D_1$. If *DL* does incur stall time, then OPT's cursor cannot pass *DL*'s cursor during the first fetch because *DL*'s holes at time t dominate OPT's holes at time t'. In this case *DL*'s cursor at time s dominates OPT's cursor at time $t' + D_1$. Thus *DL*'s cursor at time s dominates OPT's cursor at time $s' = \min\{t' + D_1, t' + D'_1\}$ and *DL*'s holes at time s dominate OPT's holes at time s' because *DL*'s holes occur at the latest possible positions. The ratio of *DL*'s segment length to OPT's segment length is at most $D_1/F \leq (d+F)/F$ because $F \leq D_1 \leq d+F$ and $D'_1 \geq F$.

It remains to analyze the case that $d'_1 > d$ and at time

$t+D_1$ there is a block in DL's cache that is referenced after the next missing block. Let $D_2 = d_2 + F$ be the number of time units after $t+D_1$ DL needs to complete the next fetch. We have $d_2 \leq d$ by the definition of DL. We distinguish two cases. (1) $d_1 + d_2 \leq d'_1$ and (2) $d_1 + d_2 = d'_1 + \delta$ for some positive integer δ. We first consider case (1). We have that DL's state at time $t + D_1$ dominates OPT's state at time $t + d'_1$. The reason is that DL's cursor at time $t + D_1$ dominates OPT's cursor at time $t+d'_1$ because OPT initiates the first fetch after t' within the next D_1 time units and DL's holes at time t dominate OPT's holes at time t', i.e. OPT's cursor cannot pass DL's cursor during the first fetch. Since OPT's holes do not change between t' and $t'+d'_1$, DL's holes at time $t + D_1$ also dominate OPT's holes at time $t + d'_1$. Since DL's state at time $t + D_1$ dominates OPT's state at time $t + d'_1$, DL's state at time $t + D_1 + d_2$ also dominates OPT's state at time $t + d'_1$ and by Lemma 1 DL's state at time $s = t + D_1 + d_2 + F = t + D_1 + D_2$ dominates OPT's state at time $s' = t' + d'_1 + F = t' + D'_1$. The ratio of the segment lengths is $(D_1 + D_2)/D'_1 \leq (d'_1 + 2F)/(d'_1 + F) \leq (d + 2F)/(d + F)$.

We next study case (2). First observe that DL's cursor at time $t+D_1+d_2$ dominates OPT's cursor at time $t'+D_1+d_2$. This is obvious if DL does not incur stall time to complete the first fetch. If DL does incur stall time, then DL's cursor at time $t+D_1$ must dominate OPT's cursor at time $t'+D'_1 \geq t' + D_1$ because DL's holes at time t dominate OPT's holes at time t' and OPT cannot finish the first fetch later in the sequence than DL. Since DL's cursor advances one step in each of the following d_2 time units after $t + D_1$ we have the stated domination for the cursors. If OPT does not initiate a second fetch before $t'+D_1+d_2$, then we are done. As in case (1) we have that DL's state at time $s = t+D_1+D_2$ dominates OPT's state at time $t' + D'_1 = t' + D_1 + d_2 - \delta$. This implies that DL's state at time s dominates OPT's state at time $s' = t' + D_1 + d_2$ because DL's cursor at time $s > t+D_1+d_2$ dominates OPT's cursor at time $t' + D_1 + d_2$ as shown above and OPT's holes do not change between $t' + D_1 + d_2 - \delta$ and s'. The ratio of the segment lengths is $(D_1+D_2)/(D_1+d_2) = (d'_1 + \delta + 2F)/(d'_1 + \delta + F) \leq (d + 2F)/(d + F)$. If OPT does initiate a second fetch before $t' + D_1 + d_2$ but at time $t + D_1 + D_2$ all k block in DL's cache are all requested before the next missing block, then DL's state at time $s = t+D_1+D_2$ dominates OPT's state at time $s' = t' + D_1 + d_2$ because DL's holes occur at the latest possible positions. The ratio of DL's segment length to OPT's segment length is upper bounded by $(D_1+D_2)/(D_1+d_2) \leq (d+2F)/(d+F)$.

We finally have to consider the case that OPT initiates a second fetch before $t' + D_1 + d_2$ but at time $t + D_1 + D_2$ there is a block DL's cache that is requested after the next missing block. DL needs $D_3 = d_3+F$ time units with $d_3 \leq d$ to complete the next fetch. Suppose that OPT initiates the second fetch at time $t' + D'_1 + \delta'$, with $\delta' \leq \delta$. As above we have that DL's state at time $t + D_1 + D_2$ dominates OPT's state at time $t' + D'_1$. This implies that DL's state at time $t + D_1 + D_2$ dominates OPT's state at time $t' + D'_1 + \delta'$ because DL's cursor at time $t + D_1 + D_2$ dominates OPT's cursor at time $t' + D'_1 + \delta \geq t' + D'_1 + \delta'$ and OPT's holes do not change between $t' + D'_1$ and $t' + D'_1 + \delta'$. It follows that DL's state at time $t + D_1 + D_2 + d_3$ dominates OPT's state at time $t' + D'_1 + \delta'$ and by Lemma 1 DL's state at time $s = t + D_1 + D_2 + d_3 + F$ dominates OPT's state at time $s' = t' + D'_1 + \delta' + F$. DL's segment length is at most $3(d+F)$ while OPT's segment length is at least $(d+2F)$. □

3. MINIMIZING STALL TIME IN PARALLEL DISK SYSTEMS

In this section we present a polynomial time algorithm for systems with D parallel disks that, given a request sequence σ, computes a prefetching/caching schedule whose stall time is at most that of an optimal solution. The schedule uses not more than $3(D-1)$ extra memory locations in cache.

The basic idea is to use the linear programming approach of [1] but to model the objective function, which measures the stall time of a schedule, in a different way. For this purpose we consider *synchronized* schedules that are defined as follows. Consider a prefetching/caching schedule for σ. A fetch operation executed from time t_1 to time t'_1 *intersects* a fetch operation performed from t_2 to t'_2 if there is a t with $t_1 \leq t \leq t'_1$ and $t_2 \leq t \leq t'_2$ but $t_1 \neq t_2$ (and hence $t'_1 \neq t'_2$). Clearly, fetch operations executed on the same disk cannot intersect. A prefetching/caching schedule is synchronized if no two fetch operations intersect. Intuitively, in a synchronized schedule fetch operations on different disks are executed completely in parallel, starting and ending at exactly the same time. For a given σ, let $s_{OPT}(\sigma)$ be the stall time of an optimal schedule for σ. We show that there exist synchronized schedules that achieve a minimum stall time provided that they may use up to $D-1$ extra cache locations.

LEMMA 2. *For any σ, there exists a synchronized schedule that achieves a stall time of at most $s_{OPT}(\sigma)$ and uses not more than $D-1$ extra memory locations in cache.*

PROOF. Let S be an optimal prefetching/caching schedule using k cache locations. We show how to modify S so that the resulting schedule is synchronized and the stall time does not increase. Suppose that (a) up to time t schedule S is synchronized and uses at most $D-1$ extra cache locations and (b) from time t on the schedule is not synchronized but uses no extra cache locations. Moreover assume that at time t a fetch operation is initiated that intersects fetches on other disks. (Initially, t is the first point in time at which a fetch operation intersecting other fetches starts.) Let t' be the time when the fetch ends. Suppose that the fetch from t to t' intersects d, $1 \leq d \leq D-1$, fetches on other disks. Let $t_1, \ldots, t_d$ be the times when these fetches start. Furthermore, let $a_1, \ldots, a_d$ be the blocks fetched and $b_1, \ldots, b_d$ be the blocks evicted during these fetch operations. The schedule is now modified as follows. We delete the fetch operations initiated at times $t_1, \ldots, t_d$ and instead fetch $a_1, \ldots, a_d$ into the $D-1$ available extra cache locations starting at time t. At time t', when these fetches end, we evict $b_1, \ldots, b_d$ from cache so that the $D-1$ extra cache locations are again available. The stall time does not increase during this modification because a possible stall time incurred at the end of the fetch at time t' was already needed for the original fetch from t to t'. At the end of the fetch blocks $b_1, \ldots, b_d$ are available for eviction because b_i was available at time $t_i \leq t'$, $1 \leq i \leq d$. From time t' on the schedule uses only k cache locations. Repeating this step for times t, $t > t'$, at which intersecting fetches are initiated, we obtain a synchronized schedule with stall time at most $s_{OPT}(\sigma)$. □

We now describe a 0-1 linear program for computing an optimal synchronized prefetching/caching schedule that uses $k+D-1$ cache locations. Let n be the number of requests in the given sequence σ. The linear program has to determine the intervals in which the synchronized fetches are performed. As in [1] we consider intervals $I=(i,j)$ of length at most F in the request sequence, $i=0,\ldots,n-1$ and $j=1,\ldots,n$. The length of an interval is $|I|=j-i-1$. Such an interval represents a fetch that starts after request r_i and ends before r_j. Since a fetch takes F time units, $F-|I|$ units of stall time are incurred at the end of I. For each such interval we introduce a variable $x(I)$ that is 1 if (synchronized) fetches are performed in interval I and 0 otherwise. The stall time of a synchronized schedule is easy to compute; it is just the sum of the stall times incurred at the end of fetch intervals. Thus we wish to minimize $\sum_I x(I)(F-|I|)$.

The rest of the linear program is similar to that given in [1], except that several constraints simplify. We say that in interval (i,j) is *properly contained* in an interval (i',j'), i.e. $(i,j)\subseteq(i',j')$, if $i\geq i'$ and $j\leq j'$. We have to ensure that at any time only one set of synchronized fetches is performed. Therefore, for any i with $1\leq i\leq n-1$ we add the constraint $\sum_{(i-1,i+1)\subseteq I} x(I)\leq 1$.

The linear program also has to determine the blocks to be fetched and evicted in each interval. We assume without loss of generality that the cache initially contains a set S_{init} of $k+D-1$ blocks from disk 1 which are never requested in σ. Let S_d be the set of blocks in σ that are stored on disk d, $1\leq d\leq D$, and let $S=S_1\cup\ldots\cup S_D\cup S_{init}$. For any interval I and any block $a\in S$ we introduce a variable $f_{I,a}$ that is 1 if a is fetched in interval I and 0 otherwise. Furthermore, for any I and any block $a\in S$ there is a variable $e_{I,a}$ that is 1 if a is evicted in I and 0 otherwise. We have to ensure that, for any interval I and any disk d, $1\leq d\leq D$, only one block from disk d is fetched. Of course such a fetch can only be performed if $x(I)=1$. Thus we add

$$\forall I,d\sum_{a\in S_d} f_{I,a}\leq x(I).$$

We also have to make sure that in each interval the number of blocks fetched is equal to the number of blocks evicted, i.e. we have

$$\forall I\sum_{a\in S} f_{I,a}=\sum_{a\in S} e_{I,a}.$$

When a request is served, the requested block must be in cache. For any $a\in S_1\cup\ldots\cup S_D$ let $i_1<i_2<\ldots<i_l$ be the indices of the requests to a. We add the constraints $\sum_{I\subseteq(0,i_1)} f_{I,a}=1$ and $\sum_{I\subseteq(0,i_1)} e_{I,a}=0$, which guarantee that a is in cache at the time of its first request. We additionally impose, for $j=1,\ldots,l-1$,

$$\sum_{I\subseteq(i_j,i_{j+1})} f_{I,a}=\sum_{I\subseteq(i_j,i_{j+1})} e_{I,a}\leq 1,$$

which implies that if a is in cache at the time of its jth reference then it is also in cache at the time of its $(j+1)$st reference. Finally we have $\sum_{I\subseteq(i_l,n)} e_{I,a}\leq 1$. Of course, a block may not be fetched or evicted when it is referenced. Thus we have, for $j=1,\ldots,l$,

$$\sum_{I:(i_j-1,i_j+1)} f_{I,a}=\sum_{I:(i_j-1,i_j+1)} e_{I,a}=0.$$

With respect to the blocks $a\in S_{init}$ we only require $\sum_{I\subseteq(0,n)} e_{I,a}\leq 1$.

We have $n\min\{F+1,n\}$ variables $x(I)$ and $O(n^2\min\{F+1,n\})$ variables $f_{I,a}$ and $e_{I,a}$. Note that we can assume $k\leq n$ since otherwise we could simply load the requested blocks into cache and then serve all requests. Also, we can assume $D\leq n$ because otherwise we just ignore the disks that do not contain a block requested in σ. Relaxing the 0-1 variables to $0\leq x(I),f_{I,a},e_{I,a}\leq 1$, we can compute in polynomial time a solution whose value is bounded by $s_{OPT}(\sigma)$. The idea of the following analysis is to show that a fractional solution to the relaxed linear program is a convex combination of polynomially many integral solutions. We can then select one of these integral solutions and achieve a minimum stall time.

Let $\mathcal{I}=\{I|x(I)>0\}$. As is [1] we can modify the fractional solution such that for any two intervals $I=(i,j)$ and $I'=(i',j')$ in $\mathcal{I}$ with $I\subseteq I'$ we have $i=i'$ or $j=j'$, i.e. intervals share a common endpoint if one is properly contained in the other. Based on this relation we can define a linear order $<$ on $\mathcal{I}$. The intervals are ordered by increasing startpoints and, if intervals have the same startpoint, they are ordered by increasing endpoints.

In order to be able to apply techniques from [1] it is crucial that in each interval $I\in\mathcal{I}$ all D disks fetch an amount of exactly $x(I)$. Clearly, there is at least one disk d with $\sum_{a\in S_d} f_{I,a}=x(I)$ since otherwise we could decrease $x(I)$. To establish this property for all I and d, we schedule dummy fetches on the idle disks in I. Since these fetches must not change the configuration of the $k+D-1$ cache locations, we introduce $D-1$ additional cache positions that initially contain $D-1$ blocks $b_1^0,\ldots,b_{D-1}^0$ from disk 1 which are never requested in σ. We then consider the intervals in $\mathcal{I}$ in the order of $<$. Let I be the jth interval considered. For any of the at most $D-1$ disks d with $\delta_d=x(I)-\sum_{a\in S_d} f_{I,a}>0$ we fetch a new block b_d^j from disk d to an extent of δ_d and evict an amount of δ_d of the blocks $b_1^l,\ldots,b_{D-1}^l$ with the smallest index l that reside in the extra $D-1$ cache locations. Blocks b_d^j, $1\leq d\leq D$ and $j>0$ are never requested in σ. The dummy blocks keep disks busy that are originally idle. It is sufficient to use at $D-1$ cache locations because, as mentioned before, in each interval there is at least one disk that fetches to an extent of $x(I)$.

We modify the optimal fractional solution even further. More specifically, it is an easy exercise to show that there is an optimal fractional solution that satisfies the following properties on the fetches and evictions. Consider the intervals in the order $<$ and let C denote the cache configuration after we have performed fetches and evictions corresponding to the first j intervals in the order. Let I be the $(j+1)$st interval.

- For any d, $1\leq d\leq D$, we fetch the block from disk d that is not completely in C and whose next reference is earliest.
- If we evict a block from disk d in I, then it is the block from disk d which is partially or completely in C and whose next reference is furthest in the future.

Based on these properties it is possible to view the prefetching/caching schedule as a process over time. For any $I\in\mathcal{I}$, define $dist(I)=\sum_{I'<I} x(I')$, i.e. $dist(I)$ is the sum of the $x(I')$ where I' precedes I in the order $<$. The time interval associated with I is $[dist(I),dist(I)+x(I)]$. Hence

there is a unique interval I associated with each time. For any interval $I \in \mathcal{I}$ and any disk d, $1 \leq d \leq D$, we sort the blocks fetched from disk d in I by increasing order of their next reference. Let $a_1, \ldots, a_l$ be the blocks in this order. Block a_i is fetched for f_{I,a_i} time units starting at time $dist(I) + \sum_{j=1}^{i-1} f_{i,a_j}$. Hence at each time instant we fetch a unique block from each disk.

As in [1], for any t in the range $[0,1)$, we construct an integral feasible solution that uses $D-1$ cache locations in addition to the $k+2(D-1)$ locations we already use. Let $\mathcal{I}_t$ be the set of intervals I in $\mathcal{I}$ associated with time instances $t_i = t+i$, for all $i \geq 0$. Each interval I in $\mathcal{I}_t$ is part of the solution for t. If $I \in \mathcal{I}_t$ is the interval associated with time t_i, then for any disk d we fetch the block that is loaded from disk d at time t_i. The algorithm for assigning evictions is slightly different from the one described in [1]. We maintain a set Q_t that is initially empty and consider the intervals in $\mathcal{I}$ in the order $<$. Let I be the current interval and $a_1, \ldots, a_l$ be the blocks evicted in I. If a_j, $1 \leq j \leq l$, is fetched back at time t_i, for some $i \geq 0$, before its next reference, then add a_j to Q_t. If $I \in \mathcal{I}$ and Q_t currently contains at least D blocks, then remove D arbitrary blocks from Q_t and evict them during I. If Q_t currently contains less than D blocks, then remove only these available blocks and evict them in I.

LEMMA 3. *For any $t \in [0,1)$, solution $\mathcal{I}_t$ is an integral feasible solution that uses a total of at most $k + 3(D-1)$ cache locations.*

PROOF. The intervals in $\mathcal{I}_t$ are disjoint. Moreover, by the definition of our algorithm for scheduling evictions, each block that is assigned to Q_t and hence evicted in an interval of $\mathcal{I}_t$ is also fetched back before its next reference in an interval of $\mathcal{I}_t$. Hence $\mathcal{I}_t$ is a feasible solution. The optimal fractional solution used to construct $\mathcal{I}_t$ uses $2(D-1)$ extra memory locations in cache. We will show that at most $D-1$ intervals in $\mathcal{I}_t$ do not have an eviction assigned. If we load the blocks fetched in those intervals into $D-1$ extra memory locations, then $\mathcal{I}_t$ is a feasible solution that uses at most $3(D-1)$ extra cache locations.

Consider our algorithm for scheduling evictions and suppose that we just finished processing interval $I \in \mathcal{I}$. For any disk d let s_d the last point in time such that disk d fetches a block that has been evicted in intervals $I' \leq I$ but not yet been fetched back at time instances corresponding to $I' \leq I$. Suppose that there exists a time before s_d such that disk d fetches a block that has not yet been evicted in intervals $I' \leq I$. Let s'_d be the earliest point in time with the property and let a_d be the block fetched at this point in time. Let b_d be any block that has been evicted in intervals $I' \leq I$ and is fetched back after s'_d. Since b_d is fetched after a_d the next reference to b_d must be after the next reference to a_d. The last eviction of b_d in intervals $I' \leq I$ must be an eviction where b_d is discarded to an extent of 1. If b_d were discarded only partially, then our optimal fractional solution would have evicted the rest of b_d in the operations where a_d is evicted because b_d's next reference is later. Thus when b_d is fetched back after s'_d it is fetched back to an extent of 1 and this fetch is performed continuously without interruption. This implies that our algorithm added b_d to Q_t. Let E_d be the total amount of evictions of blocks from disk d up to the current interval I, i.e. $E_d = \sum_{a \in S_d} \sum_{I' \leq I} e_{I',a}$. Part of this amount is fetched back continuously until time s'_d. Blocks fetched back later are, as mentioned before, fetched to an extent of 1 and added to Q_t. Hence, when the algorithm finishes processing I, $\lfloor E_d - t \rfloor + 1$ blocks from disk d have been assigned to Q_t and summing over all disks a total of $\sum_{d=1}^{D} (\lfloor E_d - t \rfloor + 1)$ blocks have been assigned to Q_t. Let $X(I) = \sum_{I' \leq I} x(I')$. When the algorithm finishes processing I, it has tried to assign $D(\lfloor X(I) - t \rfloor + 1)$ evictions because $\mathcal{I}_t$ contains $\lfloor X(I) - t \rfloor + 1$ intervals I' with $I' \leq I$ in each of which we schedule D fetches and evictions. Moreover, $X(I) = \sum_{d=1}^{D} E_d$ because in our fractional solution in each interval I' the amount of fetches and evictions is exactly $x(I')$. Hence

$$\begin{aligned} & D(\lfloor X(I) - t \rfloor + 1) - \sum_{d=1}^{D} (\lfloor E_d - t \rfloor + 1) \\ \leq \; & D(X(I) - t + 1) - \sum_{d=1}^{D} (E_d - t + 1) + D - 1 \\ = \; & D - 1. \end{aligned}$$

We conclude that at most $D-1$ fetch operations on the various disks to not get an eviction assigned. □

When constructing the solutions $\mathcal{I}_t$ as t varies from 0 to 1, we obtain a given solution not for just one value of t but for a range of values. Let $0 = x_1 < x_2 < \ldots < x_l = 1$ be the set of values such that for all t in the range $[x_i, x_{i+1})$ we obtain the same solution $\mathcal{I}_t$, $1 \leq i < l$. Hence $\mathcal{I}_{x_1}, \ldots, \mathcal{I}_{x_{l-1}}$ are the different solutions we obtain. Since each $\mathcal{I}_{x_j}$, $1 \leq j \leq l-1$, is a synchronized schedule its stall time $s(\mathcal{I}_{x_j})$ is equal to the sum of the stall times incurred by the intervals in $\mathcal{I}_{x_j}$. Giving $\mathcal{I}_{x_j}$ a weight of $x_{j+1} - x_j$, we obtain that $\sum_{j=1}^{l-1} (x_{j+1} - x_j) s(\mathcal{I}_{x_j})$ is equal to the value of the optimal fractional solution. It follows that one of the $\mathcal{I}_{x_j}$ achieves a stall time that is bounded by the value of the optimal fractional solution and hence bounded by the minimum stall time for σ. Finding such an $\mathcal{I}_{x_j}$ is easy and in fact we do not even have to compute explicitely all the $\mathcal{I}_{x_1}, \ldots, \mathcal{I}_{x_{l-1}}$. All we have to do is to compute a t_0 such that the total stall time of intervals in $\mathcal{I}_{t_0}$ is minimum among all $\mathcal{I}_t$. For varying t, the intervals in $\mathcal{I}_t$ only change if an interval $I \in \mathcal{I}$ starts at some time t_i. Thus we only have to check $|\mathcal{I}| = O(n \min\{F+1, n\})$ values of t. Once we have determined an optimal t_0, we apply our algorithm to schedule the evictions. This establishes our main result.

THEOREM 4. *There exists a polynomial time algorithm for integrated prefetching and caching on D parallel disks that, given a request sequence σ, computes a schedule whose stall time is at most that of an optimal solution for σ. The schedule uses at most $3(D-1)$ extra memory locations in cache.*

4. CONCLUSIONS

In this paper we presented improved prefetching/caching algorithms for single and parallel disk systems. In the single disk setting an interesting problem is to develop fast algorithms that achieve an even smaller approximation ratio with respect to the elapsed time performance measure. A challenging open problem is to determine the complexity of the parallel disk case: Is it NP-hard to construct optimal schedules or does there exist a polynomial time algorithm?

5. REFERENCES

[1] S. Albers, N. Garg and S. Leonardi. Minimizing tall time in single and parallel disk systems. *Journal of the ACM*, 47:969–986, 2000.

[2] S. Albers and C. Witt. Minimizing stall time in single and parallel disk systems using multicommodity network flows. *Proc. 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems APPROX)*, Springer LNCS 2129, 12–23, 2001.

[3] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[4] A. Borodin, S. Irani, P. Raghavan and B. Schieber. Competitive paging with locality of reference. *Journal on Computer and System Sciences*, 50:244–258, 1995.

[5] P. Cao, E.W. Felten, A.R. Karlin and K. Li. A study of integrated prefetching and caching strategies. *Proc. ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 188–196, 1995.

[6] P. Cao, E.W. Felten, A.R. Karlin and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transaction on Computer Systems (TOCS)*, 14:311–343, 1996.

[7] A. Fiat and M. Mendel. Truly online paging with locality of reference. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, 1997.

[8] D.R. Fuchs and D.E. Knuth. Optimal prepaging and font caching. *ACM Transactions on Programming Languages and Systems*, 7:62–79, 1985.

[9] A. Gaysinsky, A. Itai, and H. Shachnai. Strongly competitive algorithms for caching with pipelined prefetching. *Proc. 9th Annual European Symposium on Algorithms (ESA01)*, Springer LNCS 2161, 49–61, 2001.

[10] D.A. Hutchinson, P. Sanders, and J.S. Vitter. Duality between prefetching and queued writing with parallel disks. *Proc. 9th Annual European Symposium on Algorithms (ESA01)*, Springer LNCS 2161, 62–73, 2001.

[11] M. Kallahalla and P.J. Varman. Optimal prefetching and caching for parallel I/O systems. *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures*, 2001.

[12] T. Kimbrel and A.R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29:1051 – 1082, 2000. Preliminary version in FOCS96.

[13] T. Kimbrel, P. Cao, E.W. Felten, A.R. Karlin and K. Li. Integrated parallel prefetching and caching. *Proc. ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1996.

[14] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G.A. Gibson, A.R. Karlin and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. *Proc. of the ACM SIGOPS/USENIX Association Symposium on Operating System Design and Implementation*, 1996.

[15] P. Krishnan and J.S. Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27:1617-1636, 1998.

[16] M. Palmer and S.B. Zdonik. Fido: A cache that learns to fetch. *Proc. 17th International Conference on Very Large Data Bases*, 255–264, 1991.

[17] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka. Informed prefetching and caching. *Proc. 15th Symposium on Operating Systems Principles*, 79–95, 1995.

[18] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985.

[19] J. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43:771-793, 1996.