Minimizing Stall Time in Single and Parallel Disk Systems

Susanne Albers* Naveen Garg*† Stefano Leonardi ‡

Abstract

We study integrated prefetching and caching problems following the work of Cao *et. al.* [3] and Kimbrel and Karlin [13]. Cao *et. al.* and Kimbrel and Karlin gave approximation algorithms for minimizing the total elapsed time in single and parallel disk settings. The total elapsed time is the sum of the processor stall times and the length of the request sequence to be served.

We show that an optimum prefetching/caching schedule for a single disk problem can be computed in polynomial time, thereby settling an open question by Kimbrel and Karlin. For the parallel disk problem we give an approximation algorithm for minimizing stall time. Stall time is a more realistic and harder to approximate measure for this problem. All of our algorithms are based on a new approach which involves formulating the prefetching/caching problems as integer programs.

1 Introduction

Prefetching and caching are powerful tools for increasing the performance of file and data base systems. In prefetching, memory blocks are loaded from slow memory, e.g. a disk, into cache before the actual references to the blocks so as to reduce the waiting time incurred if the block were to be fetched from disk when it is referenced. Caching on the other hand tries to maintain the most frequently accessed blocks in cache so that they do not have to be fetched from disk. Both prefetching and caching have separately been the subjects of extensive theoretical and experimental studies [1, 2, 5, 6, 7, 8, 9, 14, 15, 19, 20]. However, only recently have researchers started looking at these techniques in an integrated manner and to explore interrelationships between them [3, 4, 11, 13, 16, 17]. In a seminal work Cao *et. al.* [3] introduced a model that allows an algorithmic study of the problem.

First consider the case when all blocks reside on one disk. We are given a request sequence $\sigma = r_1, \ldots, r_n$ and a cache of size k. Each of the n requests r_i specifies a memory block stored on disk. We emphasize that we study the offline problem in which the entire request sequence is given in advance. Serving a request takes one time-unit. However, a request can be served only if the block requested is in cache. Fetching a block not in cache takes F time units. Thus if we encounter a request to a block that is not in cache we can start fetching the block from disk; in this case the processor has to stall for F time-units. A better option is to initiate a fetch, *a prefetch*, to the block some i time-units before the actual reference; the processor now has to stall for only $F - i$ time-units. A

*Authors' address: Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany. {albers,naveen}@mpi-sb.mpg.de

†Supported by the EU ESPRIT LTR Project N. 20244 (ALCOM-IT).

‡This work was done while the author was visiting the Max Planck Institute für Informatik, Saarbrücken, Germany. Dipartimento di Informatica Sistemistica, Università di Roma "La Sapienza", via Salaria 113, 00198-Roma, Italia. This work was partly supported by EU ESPRIT Long term Research Project ALCOM-IT under contract n. 20244, and by Italian Ministry of Scientific Research Project 40% "Algoritmi, Modelli di Calcolo e Strutture Informative". e-mail: leon@dis.uniroma1.it.

prefetch operation may be initiated at any time provided it is the only prefetch happening at that time. However, — and this is where caching enters the picture — when we initiate a prefetch we also have to make room in cache for the in-coming block by evicting some block from cache. Thus, not only do we need to decide when to initiate a prefetch but also what blocks to fetch and evict. Starting a prefetch too early might force us to evict blocks which are requested fairly soon so that we have to initiate more prefetches to avoid stalling for these blocks. On the other hand, if a prefetch is started late, the processor might have to stall for a long time. Our goal is to minimize the total stall time, which is the total time the processor is idle. This is equivalent to minimizing the total time taken to serve the request sequence since this is just the sum of the stall times and the length of the sequence.

As an example, consider the requests sequence a, b, c, g, a, b, g, h and a cache size of 4, with blocks a, b, c and d being initially in cache. Assume $F = 5$. The minimum stall time required on this sequence is 3. On the first request to a, we start prefetching g and evict block d. Hence we have to stall for two time-units waiting for block g. On the request to g, we start prefetching h and evict c and hence have to stall for one time-unit before h is in cache.

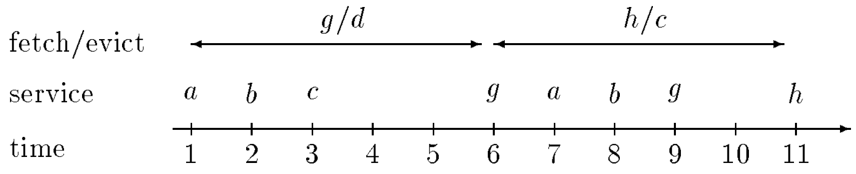

Figure 1: An example for one disk.

In the case of a parallel disk system, first explored by Kimbrel and Karlin [13], the memory blocks are distributed over D disks with each block stored on exactly one disk. At any time at most one block may be fetched from a given disk. However, blocks that reside on different disks may be prefetched in parallel. Any block in cache may be evicted to make room for a block being fetched. Thus, this corresponds to the setting where blocks are read-only and do not have to be written back to disk. Again, the goal is to minimize the total stall time. Since blocks from different disks can be fetched in parallel, an efficient strategy for the parallel disk case involves balancing the load, *ie.* the number of fetches, amongst the disks.

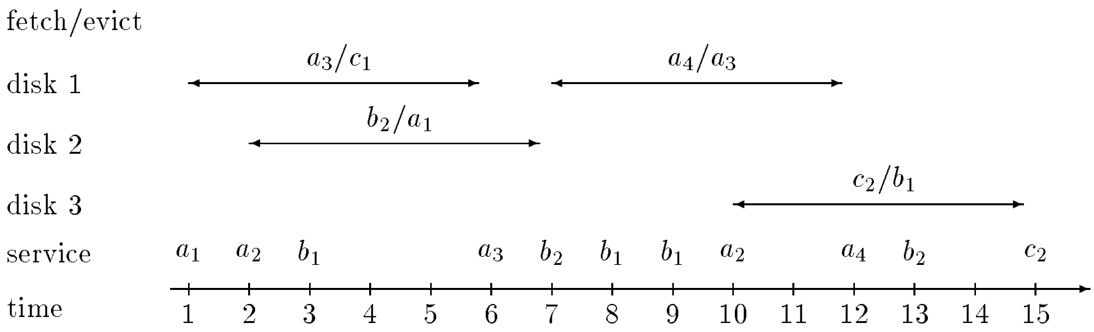

Figure 2: An example for three disks.

We give a small illustrating example for three disks. Suppose that disk 1 stores blocks a_1, a_2, a_3, a_4, disk 2 stores blocks b_1, b_2 and disk 3 stores blocks c_1, c_2. We assume $F = 5$ and a cache of size 4. Blocks a_1, a_2, b_1, c_1 are initially in cache. In Figure 2 we give a schedule for serving the request sequence

$a_1, a_2, b_1, a_3, b_2, b_1, b_1, a_2, a_4, b_2, c_2$. The total stall time is 4 time units. The schedule shows that stall time may be used simultaneously on several disks. This is the case at times 4 and 5 as well as at time 11. A disk can only prefetch blocks that are stored on it. However, evictions can be from any disk.

Previous work: Cao *et. al.* analyzed two algorithms, *conservative* and *aggressive* for the single disk problem. The *conservative* strategy incurs the same faults as Belady's optimal paging algorithm [1] while starting prefetch operations at the latest possible point in time. In contrast, the aggressive strategy starts prefetch operations at the earliest reasonable times. The elapsed time of the schedule obtained by *conservative* (respectively *aggressive*) is at most 2 (respectively $\min\{2, 1+F/k\}$) times the optimum. In addition to combinatorial analyses, Cao *et. al.* presented extensive experimental studies of the two algorithms.

Kimbrel and Karlin studied *conservative* and *aggressive* for the parallel disk problem. They showed that the approximation ratios, when the measure is the elapsed time, are $D+1$ and $D(1+(F+1)/k)$ respectively. They also presented an algorithm called *reverse aggressive*, which is the *aggressive* strategy on the reverse sequence. This algorithm achieves an approximation ratio of $(1+DF/k)$. This gives good approximation ratios if D and F/k are small, which is true in many practical applications. Karlin and Kimbrel left open the question whether an optimum prefetching/caching schedule can be computed in polynomial time even for the single disk case. A partial answer to this question was given by Kimbrel [10] who showed a dynamic programming strategy that decides whether a request sequence can be served with zero stall time in the single disk setting.

Our contribution: In this paper we present a new approach to the problem of minimizing stall time in single and parallel disk systems. We formulate the problems as integer programs and solve linear relaxations of these programs.

First, in Sections 2 and 3, we give a polynomial time algorithm for minimizing the stall time for the single disk problem, thereby settling a question left open by Kimbrel and Karlin. In particular, we show that any optimum fractional solution of our linear program can be written as a convex combination of (polynomially many) integral solutions. This is equivalent to saying that there is an optimum solution to the linear program that is integral.

All results in the mathematical programming literature that prove that the optimum solution to a certain linear program is integral do so by arguing that all vertices of the corresponding polytope are integral. This is done either by arguing that the constraint matrix is totally unimodular, as is in the case of bipartite matching and maximum s-t flow, or by combinatorial arguments as for the matching and matroid polytopes [18]. However, the polytope corresponding to the **LP** we consider has non-integral vertices. Our proof of integrality of the optimum solution exploits a certain property of the objective function we work with.

In Section 4 we study the parallel disk problem; the main novelty here being that we minimize the total stall time instead of the total elapsed time. While minimizing these two measures is equivalent, approximating total stall time is harder than approximating elapsed time, since the length of the sequence is not part of our objective function. To minimize total stall time is indeed the real objective of an efficient prefetch/caching strategy. We generalize the linear program and the proof techniques presented in Sections 2 and 3 for a single disk to the setting of parallel disks. An optimum solution to the linear program is then transformed into an integral solution that achieves an approximation ratio of D on the total stall time. The solution constructed uses at most $D-1$ additional memory locations in cache. This is actually very small – D is typically 4 or 5 – when compared with the size of the cache.

Note that for $D=1$, we obtain our optimum algorithm for the single disk case. Another pleasing feature of our algorithm is that, if a sequence can be served with zero stall time, we obtain a schedule that has no stall either and uses at most $D-1$ extra memory locations in cache. Finally, we demonstrate that if no extra memory locations are allowed, then the integrality gap of our linear program can be arbitrarily large.

In Section 5 we conclude with some remarks and open problems.

2 The LP formulation for a single disk

We assume that the request sequence is of length n. It is no loss of generality to assume that the cache is initially empty since an initial cache configuration can be modeled by prefixing the request sequence with requests to the blocks that are in cache. We identify periods in which a prefetch is performed by considering intervals of the request sequence of length at most F; the length of an interval is the number of requests in it. An interval I of length less than F is viewed as having a stall time of $F-|I|$ units at the end. With every such interval I we associate a variable $x(I)$ which is 1 if a prefetch is performed in the interval and 0 otherwise. Thus minimizing the total stall time is equivalent to minimizing $\sum_I x(I)(F-|I|)$.

To ensure that two prefetches are not performed simultaneously we add for each point r in the request sequence the constraint that $\sum_{I:r\in I} x(I) \leq 1$.

With each interval I and distinct block a we associate two non-negative variables $f_{I,a}, e_{I,a}$ which denote the extent to which block a is fetched/evicted in interval I. Clearly the total amount of fetch should be exactly equal to the total amount of eviction and this value should not exceed the value of the interval, $x(I)$. Formally,

$$\forall I \quad \sum_a f_{I,a} = \sum_a e_{I,a} \leq x(I)$$

In a feasible solution prefetches are scheduled so that a block is in cache when it is referenced. This constraint is enforced by looking at all intervals between two consecutive references to a block and requiring that on these intervals the total fetch of this block equals its total eviction which is no more than 1. Thus if the block were in cache at a certain reference it would also be in cache at the next one. Thus if i,j are two consecutive references to a block a then

$$\sum_{I\subset[i,j]} f_{I,a} = \sum_{I\subset[i,j]} e_{I,a} \leq 1$$

where $I \subset [i,j]$ denotes that interval I is properly contained[1] in the interval $[i,j]$. To ensure that every block is in cache at its first reference we require that the total fetch of a block on intervals before its first reference should be 1 and the total evict of the block on these intervals should be 0. Thus if i is the first reference to block a, $\sum_{I\subset[0,i]} f_{I,a} = 1$ and $\sum_{I\subset[0,i]} e_{I,a} = 0$.

Finally, we require that on each request, the requested block is neither prefetched nor evicted, i.e., if block a is referenced at time i

$$\sum_{I:i\in I} f_{I,a} = \sum_{I:i\in I} e_{I,a} = 0.$$

A compact description of the linear program is given in Appendix A.

Note that the only integrality constraint we imposed was on the variables $x(I)$. In any integral solution the intervals with $x(I)=1$ are non-overlapping. Given that these are the intervals in which the prefetch is to be performed, it is easy to determine the exact block to fetch/evict in each interval by using the following two rules, proposed by Cao *et. al.*, that govern optimal prefetching and eviction.

[1] Interval I is properly contained in I' if I is a subset of I' and both endpoints of I are different from those of I'.

1. *Optimal prefetching.* Fetch the block that is not in cache and is next in the stream of block references.

2. *Optimal replacement.* Evict the block from cache that is referenced latest in the future.

Our linear programming relaxation for the problem is obtained by relaxing the integrality constraint on $x(I)$ to the linear constraint $0 \leq x(I) \leq 1$. The optimum fractional solution to the linear program is an assignment of values, $x(I)$, to the intervals, I. While intervals with positive values can overlap, the sum of values of any set of pairwise overlapping intervals cannot exceed 1. Given that the prefetches need to be performed in this set of fractional intervals we can use a fractional version of the four rules to determine which blocks need to be evicted/fetched and to what extent in each interval.

3 Minimizing stall time for a single disk

In this section we consider an arbitrary optimum solution and show how to write it as a convex combination of integral solutions. It then follows that one of these integral solutions has a stall time which is at most the stall time of the fractional solution and hence at most the minimum stall time.

3.1 Modifying intervals

Let $\mathcal{I}$ be the set of intervals with $x(I) > 0$, *ie.* $\mathcal{I} = \{I | x(I) > 0\}$. An interval $I_1 = [i_1, j_1]$ is *properly contained* in interval $I_2 = [i_2, j_2]$ iff $i_1 > i_2$ and $j_1 < j_2$; a pair of intervals such that one is properly contained in the other is called a *nested-pair*. Let $I_1 \in \mathcal{I}$ be properly contained in $I_2 \in \mathcal{I}$ and let $x = \min\{x_{I_1}, x_{I_2}\}$. We reduce each of x_{I_1}, x_{I_2} by an amount x; this causes one of x_{I_1}, x_{I_2} to go down to zero and we remove the corresponding interval. We also add two new intervals $J_1 = [i_2, j_1]$ and $J_2 = [i_1, j_2]$ with $x_{J_1} = x_{J_2} = x$. The fetch in J_1 (respectively J_2) is the same as the fetch in I_1 (respectively I_2) while the evict in J_1 (respectively J_2) is the same as the evict in I_2 (respectively I_1). Since J_1 ends with I_1 the blocks that were fetched in I_1 still arrive in cache at the same time. Further, since J_1 begins with I_2 the blocks evicted in I_2 are evicted from cache at the same time as before. The same is true for the blocks fetched/evicted in interval J_2 and hence the new solution also satisfies all the constraints of the **LP**(Fig 3). Furthermore, since the total length of intervals J_1, J_2 is the same as that of I_1, I_2 and the reduction in x_{I_1}, x_{I_2} is the same as the increase in x_{J_1}, x_{J_2}, the value of the objective function remains unchanged.

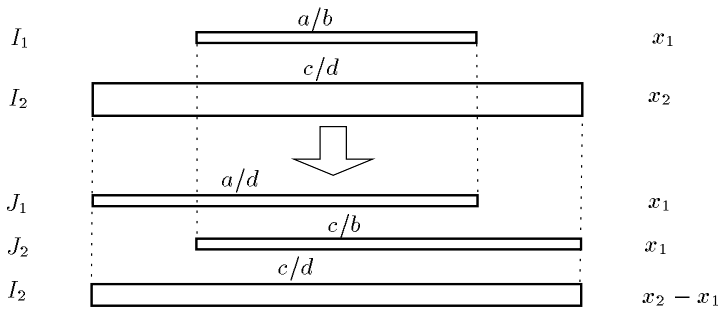

Figure 3: Eliminating nested intervals. Characters on the intervals specify "block fetched/block evicted".

Thus any nested-pair of intervals can be replaced by a set of at most 3 intervals none of which properly contains the other. By performing this transformation for every nested-pair we obtain an equivalent fractional solution without nested-pairs. Henceforth, $\mathcal{I}$ denotes this new set of intervals.

We now order the intervals in $\mathcal{I}$ by increasing starting points; if two intervals have the same start point then they are ordered by increasing end-points. We could also have ordered the intervals by increasing end-points, breaking ties by looking at starting points. It turns out that since $\mathcal{I}$ has no nested-pairs these two orderings are identical. Let $<$ denote this total order on $\mathcal{I}$.

3.2 The optimum fractional solution

As observed in [3] the optimum (integral) solution obeys the following two rules for fetching/evicting blocks: at any point the block fetched is the block not in cache whose next reference is earliest and the block evicted is that block in cache whose next reference is furthest in the future. The optimum fractional solution also follows these rules albeit in a fractional sense.

Consider intervals in the order $<$ and let C denote the cache configuration after we have performed the fetches and evicts corresponding to the first i intervals in the sequence. Note that each block is in C to an extent between 0 and 1. Further let I be the $(i+1)^{\text{th}}$ interval. There exists an optimum fractional solution for which the next two claims are satisfied.

Claim 3.1 *In I we fetch the block which is not completely in C and whose next reference is earliest.*

Proof: For contradiction assume that this block, say a, is not fetched in I and let b be one of the blocks fetched in I. We can now fetch a instead of b in interval I and fetch b in those intervals where a is fetched. Since the next reference of b is later than the next reference of a, b would be fetched before it is referenced. □

Claim 3.2 *In I we evict the block which is partially or completely in C whose next reference is furthest.*

Proof: For contradiction assume that this block, say a, is not evicted in I. Let b be one of the blocks evicted in I. We can evict a instead of b in I and fetch back a in those intervals where b is fetched. Since the next reference of a is only after the next reference of b, a would be fetched before it is referenced. □

The amount of fetch of a block prescribed by Claim 3.1 might be less than the value of I if the block is brought completely into cache. In such a case we apply the same rule to fetch another block in I. The same is true for the case of evictions in Claim 3.2. The above two claims then tell us what blocks to fetch/evict in I. This then gives us a new cache configuration which we use to decide what blocks to fetch/evict in the interval that follows I in the order $<$.

Define the *distance* of interval I, $\texttt{dist}(I)$, as the sum of the values of all intervals which precede I in $<$, *ie.*, $\texttt{dist}(I) = \sum_{\hat{I}<I} x(\hat{I})$. We can also view the process of fetching/evicting as a process in time by associating the time-interval $[\texttt{dist}(I), \texttt{dist}(I)+x(I))$ with interval I; thus there is a unique interval in $\mathcal{I}$ associated with each time-instant. We will also associate a unique fetch/evict with each time-instant. If $I \in \mathcal{I}$ is the interval associated with time t and a is the only block fetched and b the only block evicted in I then we fetch a and evict b at time t. If there are many blocks fetched/evicted in I then we order them as follows. For any two blocks a, b fetched in I, a precedes b iff the next reference to a is before the next reference to b. This defines a total order on the blocks fetched in I; let $a_1, a_2, \ldots a_i, \ldots a_p$ be the blocks in this order. Block a_i is now fetched for f_{I,a_i} time-units starting at time $\texttt{dist}(I) + \sum_{j=1}^{i-1} f_{I,a_j}$. Similarly, for any two blocks a, b evicted in I, a precedes b iff the next

reference to a is after the next reference to b. This defines another total order on the blocks evicted in I; let $b_1, b_2, \ldots b_i, \ldots b_q$ be the blocks in this order. Block b_i is now evicted for e_{I,a_i} time-units starting at time $\texttt{dist}(I) + \sum_{j=1}^{i-1} e_{I,a_j}$.

From the above two claims and our ordering of the fetches/evicts within an interval it follows that a is fetched continuously till it is fully in cache. With regard to evictions the situation is different. The eviction of a could be interrupted — before it is completely out of cache — by the eviction of another block b which is also in cache and which is better than a in the sense that its next reference is further than the next reference of a.

It will be useful to view the procedure for assigning fetches/evicts to intervals as follows. We process intervals in the order $<$ and assign fetches/evicts to them by maintaining the cache configuration and following the two rules discussed above. *Besides we also maintain a queue of those blocks which are only partially in cache; the value of a block in this queue is the extent to which it is not in cache.* Before we start evicting a block which is completely in cache we append it to the end of the queue with value 0. As we evict a block we simultaneously increase its value in the queue. If this value reaches 1, which means that the block is completely evicted, we remove it from the queue. Similarly, before we start fetching a block which is completely out of cache we add it to the front of the queue with value 1. As we fetch a block we decrease its value in the queue. When this value goes down to 0, which implies that the block is now fully in cache, we remove this block from the queue.

Lemma 3.1 *If block b is behind block a in the queue then the next reference to b is further than the next reference to a.*

Proof: The proof is by induction on the length of the queue. Suppose a is the block at the end of the queue. By the induction hypothesis the next reference to a is furthest amongst the next reference to the other blocks which are partially in cache. So if the block being evicted is only partially in cache then it is block a. As discussed above, the eviction of a could only be interrupted by the eviction of another block b whose next reference is further than the next reference of a. However, when we began evicting a its next reference was further than the next reference of b. This change in status could have happened only after a reference to block b. Hence when we started evicting b it was fully in cache and so b was appended to the end of the queue. Now b is behind a in the queue and its next reference is further than the next reference of a proving the induction claim. □

Claim 3.3 *At any point the block evicted is the block at the end of the queue.*

Proof: From the above lemma it follows that amongst blocks which are partially in cache (and hence in the queue) the block at the end of the queue is the one whose next reference is furthest. Thus the next block evicted is either this block at the end of the queue or a block which is fully in cache. In the latter case we will first append the block at the end of the queue and hence the block evicted is always the one at the end of the queue. □

Claim 3.4 *At any point the block fetched is the block at the front of the queue.*

Proof: From the above lemma it follows that if the block we fetch is partially in cache then this block is the one at the front of the queue since this is the block whose reference is earliest from amongst the blocks in the queue. Else we fetch a block that is completely out of cache that is first added to the front of the queue. □

In the remainder of this subsection we consider the fetches/evictions of a block a between two consecutive references to a.

Lemma 3.2 *Every interruption in the eviction of a is for some integral time-units.*

Proof: Once the eviction of a is interrupted it is resumed only when all blocks that were appended to the queue after a are completely evicted. Hence the total length of the interruption in the eviction of a is integral. □

We say that a is *partially fetched/evicted* if the total extent to which a is fetched/evicted between these two consecutive references is strictly less than one.

Lemma 3.3 *If a is partially fetched/evicted, then the fetch of a begins an integral time-units after the start of its evict.*

Proof: Since the value of a block in the queue is the extent to which the block is not in cache it follows that at any point the sum of the values of the blocks in the queue is integral. In particular, this is also true for the time at which we start evicting a; let the sum at this time be p. Since a is not evicted fully, all blocks that were in the queue when a was appended are not evicted further. We start fetching a only after we have fetched back all these blocks. Since the total value of these blocks is p it takes p time-units to fetch all these blocks back. The other blocks fetched are completely out of cache and so they are fetched for a unit-time each. Thus the total time between the start of the evict and the start of the subsequent fetch to a is integral. □

Lemma 3.4 *If a is evicted at time t, then there is a time $t' = t + i$, for some integer i, at which a is fetched back.*

Proof: We first assume that a is partially fetched/evicted. By Lemma 3.3 the difference in the times at which we start evicting a and fetching a back is integral. Once we start fetching a we fetch it continuously till it is completely in cache. The eviction of a could however be interrupted. But by Lemma 3.2 every interruption is for an integral time-unit. These facts together imply the lemma.

If a is fetched/evicted completely then it is no more the case that the start of the eviction and the fetch of a are integral time-units apart. However, it is still true that once we begin fetching a we fetch it continuously for one time-unit after which it is completely in cache and that every interruption in the eviction of a is for an integral time-unit. These two facts again imply the lemma. □

3.3 The convex decomposition

Claim 3.5 *Let t_1, t_2 be two time-instants such that $t_2 = t_1 + i$ for some positive integer i, and let I_1, I_2 be the intervals associated with these time-instants. Then I_1 and I_2 are disjoint.*

Proof: We have $t_2 \geq t_1 + 1$. Therefore the sum of the values of all intervals between (and including) I_1 and I_2 in $<$ is at least 1. Hence I_1, I_2 cannot overlap. □

We decompose the fractional solution into a convex combination of integral solution as follows. Let t be in the range $[0, 1)$ and let $t_i = i + t$ for every integer i, $0 \leq i \leq n$. Let $\mathcal{I}_t$ be the intervals corresponding to the time-instants t_i; by Claim 3.5 these intervals are disjoint. In the interval corresponding to t_i we schedule the fetch/evict associated with t_i. By Lemma 3.4 the set of intervals $\mathcal{I}_t$ together with this schedule of fetches and evicts forms an integral solution to the problem.

Consider the different solutions obtained as t varies from 0 to 1. Note that each solution is obtained not for just one value of t but for a range of values, say for all t in the range $[a, b]$. We assign this solution a weight $b - a$ in the decomposition. Clearly, the total weight of the solutions that an interval

I occurs in equals $x(I)$. Further, since t ranges from 0 to 1, the sum of the weights assigned to all solutions is 1. Hence, this collection of solutions with the associated weights is a convex decomposition of the optimum fractional solution.

4 The multiple disk case

In this setting the blocks are distributed over D different disks. At any point we can fetch at most one block from a disk but fetches from different disks may proceed simultaneously.

4.1 The linear program

The linear program for this case differs from the one for the single-disk setting in that we now have one copy of interval I for each disk. Let I^d, $d = 1, \ldots, D$, denote the copy of interval I for disk d; henceforth we view intervals $I^1, I^2, \ldots, I^D$ as distinct intervals. Let $x(I^d)$ be the value of interval I^d and let $e_{I^d,a}$, $f_{I^d,a}$ be the extent to which block a is evicted, fetched in interval I^d. Since only blocks that reside on disk d can be fetched in interval I^d we have that $f_{I^d,a} = 0$ if a is not on disk d. As before

$$\forall I^d \quad \sum_a f_{I^d,a} = \sum_a e_{I^d,a} \leq x(I^d).$$

To ensure that prefetches to a disk are not performed simultaneously we add for each point i in the request sequence and for each disk d, $1 \leq d \leq D$, the constraint $\sum_{I^d : i \in I^d} x(I^d) \leq 1$. As in the single disk setting we require that the total fetch of a block a on intervals between two consecutive references of a equals the total eviction of a on these intervals and is at most 1. Moreover, no block may be fetched or evicted while it is referenced.

Let $\mathcal{I}$ be the set of intervals in an integral solution to this linear program, *ie* those intervals with $x(I) = 1$. Then the stall time for this solution is at least $\left(\sum_{I \in \mathcal{I}} F - |I|\right)/D$. Hence the objective function for this linear program is to minimize $\left(\sum_I x(I)(F - |I|)\right)/D$. We will construct an integral solution with stall time at most $\sum_I x(I)(F - |I|)$, which is at most D times the optimum. In Appendix B we give an alternative linear program that models the objective function more accurately. However, we show that the approximation ratio achieved using the corresponding linear program relaxation cannot be better than D.

4.2 The optimum fractional solution

Let $\mathcal{I}^d = \left\{ I^d | x(I^d) > 0 \right\}$ be the set of intervals from disk d which have a positive value and let $\mathcal{I} = \cup_d \mathcal{I}^d$. As in the single disk setting we can modify intervals so that $\mathcal{I}^d$ contains no nested-pairs. We order intervals in $\mathcal{I}$ by increasing starting points with ties broken first by increasing ending points and then by the number of the disk to which the interval belongs; let $<$ denote this order. Note that for intervals from one disk the order $<$ is exactly the same as for the single-disk setting.

Once again consider intervals in the order $<$ and let C denote the cache configuration after we have performed fetches and evicts corresponding to the first i intervals in this order. Let I^d be the $(i+1)^{\text{th}}$ interval.

Claim 4.1 *In I^d we fetch the block from disk d which is not completely in C and whose next reference is earliest.*

Claim 4.2 *If we evict a block from disk j in interval I^d then this is that block from disk j which is partially or completely in C and whose next reference is furthest.*

4.3 Constructing an integral solution

The multi-disk setting therefore differs from that of the single-disk in that for an interval I^d we only know what block to evict from each disk; we do not know the relative amounts of the evictions of blocks from different disks.

As in the single-disk setting define the distance of an interval I^d, $\texttt{dist}(I^d)$, as the sum of the values of intervals in $\mathcal{I}^d$ which precede I^d in the order $<$, *ie.*, $\texttt{dist}(I^d) = \sum_{\hat{I}^d < I^d} x(\hat{I}^d)$. Once again we view this as a process in time by associating the time-interval $[\texttt{dist}(I^d), \texttt{dist}(I^d) + x(I^d)]$ with interval I^d. Thus there is a unique interval in $\mathcal{I}^d$ associated with each time-instant. As before we order the blocks fetched in I^d by increasing order of their next references. Let $a_1, a_2, \ldots a_p$ be the blocks in this order. Block a_i is now fetched for f_{I^d,a_i} time-units starting at time $\texttt{dist}(I^d) + \sum_{j=1}^{i-1} f_{I^d,a_j}$. Thus at each time-instant we fetch a unique block from each disk.

At each time-instant we will also evict a unique block from each disk. Let $\mathcal{P}^d$ be the set of blocks that reside on disk d. Let $a_1, \ldots, a_p \in \mathcal{P}^d$ be the blocks from disk d that are evicted in interval I ordered in decreasing order of their next reference. Block a_i is evicted for e_{I,a_i} time-units starting at time $\sum_{\hat{I}<I, \hat{a}\in\mathcal{P}^d} e_{\hat{I},\hat{a}} + \sum_{j=1}^{i-1} e_{I,a_j}$. Note that if there was only one disk then the time at which we start evicting a_i is exactly the same as $\texttt{dist}(I) + \sum_{j=1}^{i-1} e_{I,a_j}$ which was how we had defined the starting time of this eviction earlier. However, if a_i is evicted at time t then, unlike the single-disk setting, it is not necessary that in the fractional solution a_i is evicted in one of the intervals associated with time-instant t.

The machinery we developed for the single-disk case can now be applied to each disk in the multi-disk setting. A queue is associated with each disk d. We consider the fetches/evictions of blocks that reside on this disk as a process in time and update the queue as in the single-disk case. Using Claims 4.1 and 4.2 we can extend Lemma 3.1 from which Claims 3.3 and 3.4 follow. It is also straightforward to extend Lemmas 3.2 and 3.3 which can then be used, exactly as before, to prove Lemma 3.4 for the multi-disk setting.

Extending Claim 3.5 to the multi-disk setting yields

Claim 4.3 *Let t_1, t_2 be two time-instants such that $t_2 = t_i + i$ for some positive integer i, and let I_1^d, I_2^d be the intervals on disk d associated with these time-instants. Then I_1^d and I_2^d are disjoint.*

We now show how to obtain an integral solution. Let t be in the range $[0, 1)$ and let $t_i = i + t$ for every integer i, $0 \leq i \leq n$. To each time-instant t_i and disk d there corresponds an interval; our solution contains all these intervals and let $\mathcal{I}_t$ denote this set of intervals. In the interval corresponding to t_i and disk d we fetch the block from disk d that is fetched at time t_i. The block that resides on disk d and is evicted at time t_i will also be evicted in this solution, albeit in a different interval.

Evictions are assigned to intervals of $\mathcal{I}_t$ in the following manner. Consider the intervals in $\mathcal{I}$ in the order $<$ and let I be the current interval. Suppose there is a block that is evicted in I and the same eviction is scheduled at time t_i for some i. We then add this block to a set S (S is the set of evictions that need to be assigned to intervals of $\mathcal{I}_t$ and is initially empty). If $I \in \mathcal{I}_t$ and S is not empty then remove a block from S and assign it to interval I; no block is evicted in interval I in this solution if the set S is empty.

By Claim 3.5 any two intervals in $\mathcal{I}_t$ that are from the same disk are disjoint. If in our solution we fetch a block in an interval I then the same block is fetched in I in the fractional solution. If the fractional solution evicts a block in an interval I then in our solution the block is evicted in an interval whose starting point is only after the starting point of I. Next consider two consecutive references to a block a. By Lemma 3.4 it follows that if a is evicted in some interval of this solution then it is also fetched back. Thus this assignment of fetches/evictions to intervals of $\mathcal{I}_t$ is a feasible solution to the problem provided every interval of $\mathcal{I}_t$ has an eviction assigned to it. We next prove that at most $D-1$ intervals do not have an eviction assigned.

Lemma 4.1 *For any t there are at most $D-1$ intervals in $\mathcal{I}_t$ that do not have an eviction assigned.*

Proof: Our procedure for assigning evictions to intervals of $\mathcal{I}_t$ considers intervals of $\mathcal{I}$ in the order $<$. At any step let F be the number of intervals of $\mathcal{I}_t$ encountered and E the number of evictions encountered that are to be assigned to intervals in $\mathcal{I}_t$. We first prove that $F - E \leq D - 1$.

Let f_d, e_d be the total amount of fetch, evict of blocks from disk d till this point. Clearly, $\sum_{d=1}^{D} f_d = \sum_{d=1}^{D} e_d$. Further, $F = \sum_{d=1}^{D} \lfloor f_d - t + 1 \rfloor$ and $E = \sum_{d=1}^{D} \lfloor e_d - t + 1 \rfloor$. The claim that $F \leq E + D - 1$ follows from

$$F = \sum_{d=1}^{D} \lfloor f_d - t + 1 \rfloor \leq \lfloor \sum_{d=1}^{D} (f_d - t + 1) \rfloor = \lfloor \sum_{d=1}^{d} (e_d - t + 1) \rfloor < \sum_{d=1}^{D} \lfloor e_d - t + 1 \rfloor + D = E + D.$$

Assume that the interval I is in $\mathcal{I}_t$ and there are $D-1$ intervals preceding I in order $<$ that belong to $\mathcal{I}_t$ and do not have an eviction assigned. Since at any point $F - E \leq D - 1$, the set S is not empty and hence I will be assigned an eviction. □

Since at most $D-1$ intervals do not have an eviction assigned, we can use $D-1$ extra cache locations to fetch the blocks fetched in these intervals. Note that a block fetched into one of these extra locations can be evicted later and replaced by a different block. Thus for every $t \in [0, 1)$ we have a feasible solution that uses at most $D-1$ extra blocks in cache.

Consider the different solutions obtained as t varies from 0 to 1. Note that each solution is obtained not for just one value of t but for a range of values. Let $0 \leq x_1 < x_2 < \ldots < x_s < 1$ be a set of values such that if we start fetching/evicting a block at time t on disk d or if $\texttt{dist}(I^d) = t$ for some I^d then there exists a value x_i such that $x_i = t \bmod 1$. From our definition of $\mathcal{I}_t$ and the fetches/evictions assigned to intervals in $\mathcal{I}_t$ it follows that if $x_i \leq t < x_{i+1}$ then we would obtain the same solution for all values of t in the range $[x_i, x_{i+1})$. We assign this solution a weight $x_{i+1} - x_i$. Clearly, the total weight of the solutions that an interval I^d occurs is equals $x(I^d)$. Further, since t ranges from 0 to 1, the sum of the weights assigned to all solutions is 1. Hence, this collection of solutions with the associated weights is a convex decomposition of the optimum fractional solution.

We would like to select the best among the s integral solutions. The number of solutions we construct is bounded by the total number of fetches/evictions of blocks over all the intervals in the fractional solution. This number is bounded by $O(DFn^2)$.

We can therefore conclude with the following theorem.

Theorem 4.2 *There exists a polynomial time algorithm for the prefetch/caching problem on D parallel disks, that produces a solution with at most D times the optimum stall time using at most $D-1$ extra memory locations.*

Observe that for $D = 1$, we get a solution with minimum stall time without using any extra memory locations. In Appendix B we show that if no extra memory locations are used, then the integrality gap of our linear program can be arbitrarily large.

5 Conclusion

In this paper we presented a polynomial time algorithm for optimal prefetching/caching on a single disk. For the parallel disk problem we developed a D-approximation algorithm that is allowed to use $D-1$ extra memory locations in cache.

We can remove the additional memory locations at the expense of increasing the stall time. The intergral solution constructed in Section 4.3 works on a cache of size k. Consider one of the $D-1$ prefetch operations that do not have an eviction assigned. In this operation we now evict the block a in cache whose next reference is furthest in the future. If a is evicted in some other interval I before the next reference to a, then we cancel the eviction there; otherwise we introduce an interval I right before the reference to a and fetch a. In any of the two cases, the block to be evicted in I is determined in the same way as before. We repeat this process until the end of the request sequence is reached. In the same way we process the other $D-2$ prefetch operations that do not have an eviction assigned. We obtain a schedule in which every prefetch operation has an eviction assigned. The extra stall time introduced is at most $(D-1)\frac{F}{k}n$. The total elapsed time is bounded by $(1+(D-1)\frac{F}{k})n+Ds$, where n is the length of the request sequence and s is the stall time before the application of the procedure. The approximation of the elapsed time so obtained improves over the factor $(1+D\frac{F}{k})$ of the algorithm by Kimbrel and Karlin if $\frac{F}{k} \geq 1$.

An interesting open problem is to find a combinatorial, polynomial time algorithm for minimizing stall on a single disk. A challenging open problem is to find a constant approximation algorithm for the parallel disk problem.

References

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

[2] 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.

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

[4] K.M. Curewith, P. Krishnan and J.S. Vitter. Practical prefetching via data compression. In *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, pages 257–266, 1993.

[5] C.S. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, pages I306–314, 1989.

[6] A. Fiat, R.M. Karp, L.A. McGeoch, D.D. Sleator and N.E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[7] A. Fiat and A.R. Karlin. Randomized and multiprocessor paging with locality of reference. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 626–634, 1995.

[8] A. Fiat and Z. Rosen. Experimental studies of access graph based heuristics. Beating the LRU standard. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 63–72, 1997.

[9] A.R. Karlin, S. Phillips and P. Raghavan. Markov paging. *Proc. 33nd Annual Symposium on Foundations of Computer Science*, pages 208–217, 1992.

[10] T. Kimbrel. Zero-stall prefetching and caching. Manuscript 1997.

[11] T. Kimbrel, P. Cao, E.W. Felten, A. Karlin and K. Li. Integrated parallel prefetching and caching. In *Proc. ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 262–263, 1996.

[12] D. Kotz and C.S. Ellis. Practical prefetching techniques for mulitprocessor file systems. *Journal of Distributed and Parallel Databases*, 1:33–51, 1993.

[13] T. Kimbrel and A.R. Karlin. Near-optimal parallel prefetching anch caching. In *Proc. 37th IEEE Annual Symposium on Foundations of Computer Science*, pages 540–549, 1996.

[14] P. Krishnan and J.S. Vitter. Optimal prediction for prefetching in the worst case. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 392–401, 1994.

[15] K.-K. Lee and P. Varman. Prefetching and I/O parallelism in multiple disk systems. In *Proc. 1995 International Conference on Parallel Processing*, pages III160–163, 1995.

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

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

[18] A. Schrijver. *Linear and Integer Programming.* Wiley, Chichester, 1986.

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

[20] J. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proc. 32nd Annual Symposium on Foundations of Computer Science*, pages 121–130, 1991.

Appendix A

We give the linear program for minimizing stall time for a single disk and need one more definition. In the request sequence σ, let $a_1, \ldots, a_{n_a}$ be the requests to block a.

$$
\begin{array}{llcll}
\text{minimize} & \sum_I x(I)(F-|I|) & & & \\
\text{subject to} & & & & \\
& \displaystyle\sum_{I: r \in I} x(I) & \leq & 1 & \forall r \\
& \displaystyle\sum_a f_{I,a} = \sum_a e_{I,a} & \leq & x(I) & \forall I \\
& \displaystyle\sum_{I \subset [a_i, a_{i+1}]} f_{I,a} = \sum_{I \subset [a_i, a_{i+1}]} e_{I,a} & \leq & 1 & \forall a, i \\
& \displaystyle\sum_{I \subset [0, a_1]} f_{I,a} & = & 1 & \forall a \\
& \displaystyle\sum_{I \subset [0, a_1]} e_{I,a} & = & 0 & \forall a \\
& \displaystyle\sum_{I: a_i \in I} f_{I,a} = \sum_{I: a_i \in I} e_{I,a} & = & 0 & \forall a, i \\
& x(I) & \in & \{0,1\} & \forall I
\end{array}
$$

Appendix B

An alternative **LP** formulation for minimizing stall time in the multi-disk setting would be as follows. We have stall-variables s_i indicating the extent of the stall just before the i^{th} request is served. Thus the objective would now be to minimize $\sum_{i=1}^{n} s_i$. Once again we have a variable $x(I^d)$ associated with the copy of interval I on disk d where I is of length at most F. We also have fetch and evict variables associated with every 3-tuple, (page, interval, disk), as before. All constraints from the earlier **LP** still apply. However, we now need additional constraints to ensure that for every interval I that is chosen the sum of the stall times before the requests in this interval is at least $F - |I|$. It will be convenient to have a variable $s_{i,d,I}$ indicating the stall time before the i^{th} request when a block was fetched from disk d in interval I. Then for every disk d and interval $I = [p, q]$ we have

$$\sum_{i=p}^{q} s_{i,d,I} \leq x(I^d)(F - |I|).$$

Let $s_{i,d}$ be the stall before the i^{th} request due to a block that was being fetched from disk d. Then

$$s_{i,d} = \sum_{I: i \in I} s_{i,d,I}.$$

Now the stall time before the i^{th} request is the maximum of the times spent waiting for blocks that were fetched from different disks and hence $s_i = \max_d s_{i,d}$. Since the objective is to minimize the sum of the stall times s_i, we need the set of inequalities

$$s_i \geq s_{i,d} \quad 1 \leq d \leq D.$$

In this linear program we relax again the integrality constraint on $x(I)$ to the linear constraint $0 \leq x(I) \leq 1$. Using this relaxation, we cannot achieve an approximation ratio on the stall time that is better than D. Consider a cache of size $D+1$, with blocks $a_1, c_1, \ldots, c_D$ being initially in cache. Block a_1 is stored on disk 1 and block c_i, for $1 \leq i \leq D$, is stored on disk i. The request sequence to be served is $(a_1)^F, b_1, c_1, \ldots, c_D$ where block b_1 is stored on disk 1. Here $(a_1)^F$ represents F requests to a_1.

An optimum fractional solution for serving this sequence prefetches b_1 during the F requests to a_1 and evicts every block c_i, $1 \leq i \leq D$ to an extent of $1/D$. Starting with the request to b_1, the D disks simultaneously fetch the missing portions of $c_1, \ldots, c_D$. Before the request to c_1 a stall of $F-1$ time units has to be introduced. However, since each disk only prefetches a block to an extent of $1/D$, $s_{F+1} = s_{F+1,d}$ for all d and thus the objective function value is $\frac{1}{D}(F-1)$.

An optimum integral solution, when prefetching b_1, evicts block c_D. On the request to b_1, disk D starts prefetching c_D while the other disks are idle. Before request c_D, a stall time of $F-D$ time units has to be inserted. This gives a performance ratio of $(F-D)/(\frac{1}{D}(F-1)) = D(1-\frac{D-1}{F-1})$, which can be arbitrarily close to D.

Consider the intergral solution constructed in Section 4.3. We show that if no extra memory blocks are allowed, the integrality gap of our linear program can be arbitrarily large. This holds even for problems on two disks. We give a request sequence σ such that (a) there exists a fractional solution with zero stall time and (b) there exists no integral solution with zero stall time.

The request sequence σ is composed of three subsequences σ_1, σ_2 and σ_3. We first give zero stall time solutions for $\sigma_{12} = \sigma_1\sigma_2$ and $\sigma_{23} = \sigma_2\sigma_3$ and then show that there is no integral solution for $\sigma = \sigma_1\sigma_2\sigma_3$ that has zero stall time.

Consider a system with two disks. We need 12 blocks a_i, b_i, c_i and d_i, $1 \leq i \leq 3$, where a_i and c_i are stored on disk 1 and b_i and d_i are stored on disk 2, $1 \leq i \leq 3$. Let

$$\sigma_1 = b_1, b_2, b_2, b_2, a_1, a_2, c_3, d_3, c_1, c_2, d_1, d_2, c_1, a_3, b_3, c_2, c_1, a_1$$

$$\sigma_2 = d_1, d_2, b_1, a_2, b_2$$

$$\sigma_3 = c_1, d_2, c_2, a_3, b_3, a_1, a_2, b_1, b_2, c_3, a_1, d_3, c_1, a_2, b_1, b_2, a_1, c_2, d_1, d_2.$$

We assume a cache of size 10, where initially all but blocks a_3 and b_3 are stored in cache. The stall time is $F = 8$.

Figure 4 shows zero stall time schedules for the sequence $\sigma_{12} = \sigma_1\sigma_2$. The intervals above the request sequence represent an optimum fractional solution, where each interval I has an associated value $x(I) = 1/2$. The intervals below the request sequence represent the integral solution in which fetches on disk 1 are completed as early as possible. An earlier completion time on disk 1 could only be achieved if, in the first prefetch operations, disk 1 evicts b_1 and disk 2 evicts b_2. However, this leads to a schedule with non-zero stall time because disk 2 cannot simultaneously prefetch b_1 and b_2. Note that at the end of the schedules, blocks c_3 and d_3 are not in cache.

Figure 5 shows solutions for the request sequence $\sigma_{23} = \sigma_2\sigma_3$ given an initial cache in which blocks c_3 and d_3 are missing. The integral solution given below the request sequence is the only integral solution with zero stall time. In an integral solution, disk 1 must evict d_1 in the first prefetch operation. It is impossible to evict c_1 because c_1 cannot be fetched back in time. Given that disk 1 evicts d_1, disk 2 must evict c_1 in its first prefetch operation; otherwise d_1 cannot be fetched back in time. This requires that the prefetch on disk 1 starts on request d_2.

For the sequence $\sigma = \sigma_1\sigma_2\sigma_3$, the fractional solutions in Figure 4 and 5 can be combined and give an optimum fractional solution for σ. However, there is no integral solution with zero stall time. To serve $\sigma_1\sigma_2$, disk 1 must prefetch a_2 while serving request d_2 in σ_2. To serve $\sigma_2\sigma_3$, disk 2 must prefetch c_1 while serving that particular request.

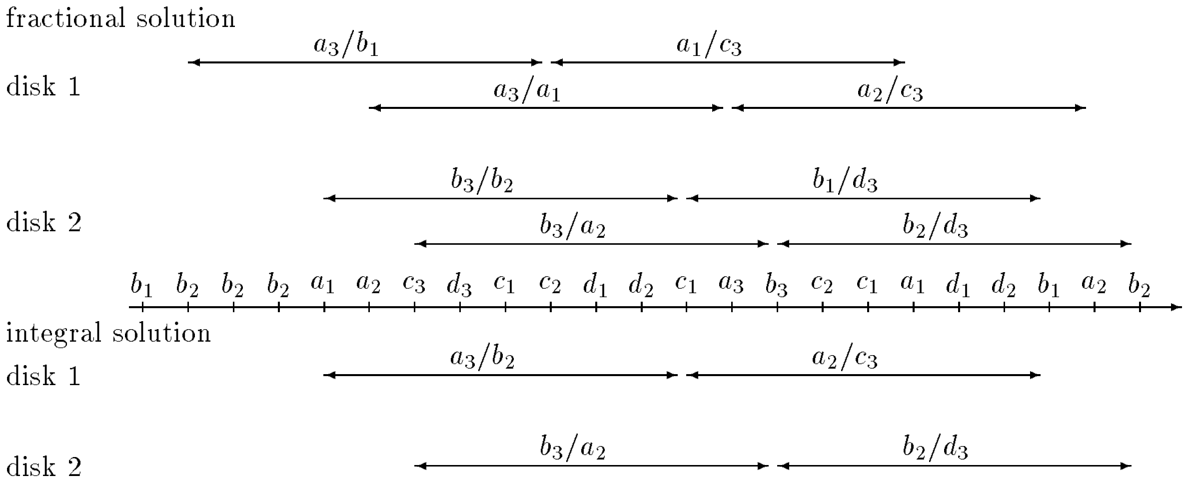

Figure 4: Fractional and integral solutions for the sequence $\sigma_{12} = \sigma_1\sigma_2$.

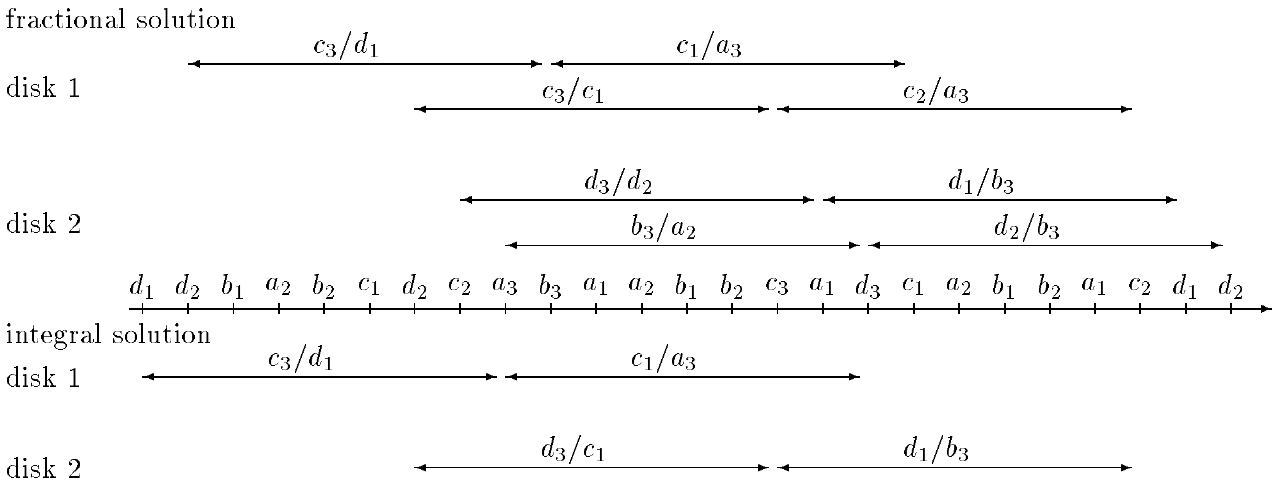

Figure 5: Fractional and integral solutions for the sequence $\sigma_{23} = \sigma_2\sigma_3$.