Near-optimal parallel prefetching and caching

Tracy Kimbrel Anna R. Karlin
Department of Computer Science and Engineering
University of Washington
Seattle, Washington
{tracyk,karlin}@cs.washington.edu

Abstract

Recently there has been a great deal of interest in the operating systems research community in prefetching and caching data from parallel disks, as a technique for enabling serial applications to improve I/O performance. [16, 30, 32, 41, 51, 42]. We consider algorithms for integrated prefetching and caching in a model with a fixed-size cache and any number of backing storage devices (which we will call disks). The integration of caching and prefetching with a single disk was previously considered by Cao et al. [8]. We show that the natural extension of their *aggressive* algorithm to the parallel disk case is suboptimal by a factor of (nearly) the number of disks in the worst case. Our main result is a new algorithm, *reverse aggressive*, with near-optimal performance for integrated prefetching and caching in the presence of multiple disks.

1 Introduction

1.1 Motivation

Recent advances in technology have made magnetic disks both cheaper and smaller. As a result, parallel disk arrays have become an attractive means for achieving high performance from storage devices at low cost. Multiple disks offer the advantages of both increased bandwidth and reduced contention. Nonetheless, there are many applications which do not benefit from this I/O parallelism as much as they could, and end up stalling for I/O a significant fraction of the time.

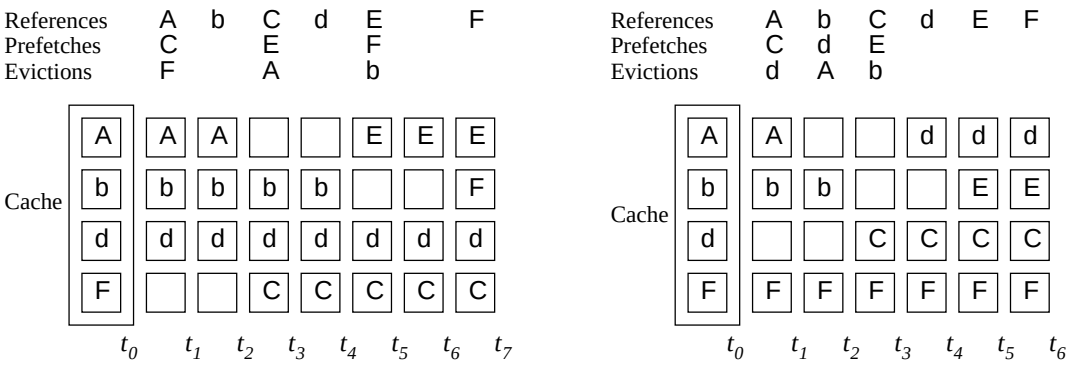
At the same time, it has been observed that many of these applications have largely predictable access patterns. This has enabled the use of prefetching and informed cache replacement(e.g., [16, 30, 32, 41, 51, 42]) as techniques for reducing I/O overhead in such systems. The two techniques are not independent, however, and can interact poorly if their interaction is not considered carefully [8, 41].

In this paper, we consider a theoretical model that captures the important characteristics of a system for prefetching and caching with multiple disks. We study the offline problem of constructing an optimal prefetching and caching schedule in this model, for a given request stream. An optimal schedule minimizes the elapsed time required to serve the given request stream. Although complete information about future requests is usually not available, partial information *is* often available in the form of limited or even significant lookahead into the request stream. Empirically we have found that a limited-lookahead version of our algorithm outperforms other approaches in practice [28]. In addition, the design and analysis of the optimal offline algorithm is an important step towards understanding and evaluating more practical limited-lookahead algorithms.[1]

Surprisingly, even in the offline, single-disk situation, this is a challenging combinatorial problem: we know of no polynomial time algorithm for determining an optimal prefetching schedule. The difficulty comes from the fact that prefetching too soon can cause additional cache misses by replacing blocks that would remain in the cache if prefetching were done later or not at all: new and possibly better eviction opportunities arise as a program proceeds. Nonetheless, Cao et al. [8] were able to show that a simple and natural algorithm called *aggressive*, which prefetches as early as is reasonable, has performance that is provably close to optimal in the single disk case.

We show, however, that the natural extension of this algorithm to the multiple disk case has performance that is suboptimal by a factor nearly equal to the number of disks. The interaction between caching and prefetching is significantly more complicated in a system with multiple disks because a set of blocks can be prefetched in parallel only if they reside on different disks: each disk can serve only one prefetch at a time. The prefetching schedule and choice of cache evictions impact the poten-

[1] We can perhaps draw an analogy with the impact of the optimal offline paging algorithm [1] on the design, implementation and evaluation of online paging algorithms.



(a) A natural extension to the aggressive algorithm.

(b) A better algorithm.

Figure 1: An example of prefetching and caching with two disks. One disk holds blocks A, C, E, and F, and another disk holds blocks b and d. The cache size is $K = 4$ and the fetch time is $F = 2$.

tial for subsequent parallel prefetching in a complex way. Our main result is a new algorithm, *reverse aggressive*, with near-optimal performance for this problem.

1.2 An Example

An example will serve to introduce our model and illustrate the challenge posed by the multi-disk problem. In the example, the cache holds four blocks. The application references one block per time unit. If the application wants to reference a block that is not present in the cache, the application must wait or *stall* until the block is present. In this example, it takes two time units to fetch a block from disk, and each disk can perform only one fetch at a time. Every fetch evicts some block from the cache; the evicted block becomes unavailable at the moment the fetch starts. The goal is to minimize the total time spent by the application, or equivalently to minimize the stall time.

The application references blocks according to the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d, and F. Blocks A, C, E, and F reside on one disk; blocks b and d on a different disk. A straightforward approach is to use the *aggressive* algorithm [8]: always fetch the missing block that will be referenced soonest; evict the block whose next reference is furthest in the future; but don't fetch if the evicted block will be referenced before the fetched block. Figure 1(a) shows the cache block changes. For example, initially, the first missing block is C, and the block whose next reference is furthest in the future is F. Moreover, the reference to F is after the reference to C. Therefore, the *aggressive* algorithm immediately initiates a fetch for C, evicting F. Notice that this fetch is entirely overlapped wtih computation (the references to A and b). The schedule produced using this algorithm results in one unit of stall time (the sixth time unit). The entire sequence is served in seven time units.

Figure 1(b) shows another schedule that is faster by one time unit. On the first fetch, d is evicted rather than F, even though d is referenced earlier. This has the advantage of offloading one fetch from the heavily loaded disk to the otherwise idle disk. This change allows two fetches to proceed in parallel later, thus saving one time unit.

The example shows that it is helpful to take disk load into account when making fetching and eviction decisions. This is the factor that makes the multi-disk problem more difficult than the single-disk problem.

1.3 Overview of Results

Our model generalizes the previous example in the obvious way. Its parameters are as follows.

- Let d be the number of disks.
- Let B be a set of blocks. We will refer to the disk on which a block $b \in B$ resides as the *color* of b.
- There is a cache that contains at most K blocks in B at any time.
- A *reference sequence*, or *request sequence*, is an ordered sequence of references $R = r_1, r_2, \ldots r_{|R|}$, where each $r_i \in B$.
- Fetching a block from a disk into the cache takes F time units.

We imagine that there is a *cursor* which at any time points to the next request to be served. If this request is for a block that is in the cache, the cursor advances by one during the next time unit. If this request is for a block that is not in the cache, the cursor *stalls* until that block arrives in the cache (i.e., until the fetch

for that block completes). Note that to the extent that the cursor is advancing, a prefetch can overlap the serving of requests. Also, prefetches can overlap each other provided that the prefetched blocks reside on different disks.

We consider three algorithms for parallel prefetching in this paper, *conservative*, *aggressive* and *reverse aggressive*. The first two are natural extensions of the two single disk prefetching strategies described in [8]. They lie at opposite ends of the spectrum in terms of the total number of fetches performed: *Conservative* performs the minimum possible number of fetches, at the expense of a worse elapsed time in the worst case; *Aggressive* prefetches as aggressively as possible without making things worse.

We give nearly tight bounds on the performance of both of these algorithms. Unfortunately, for both of these algorithms, there are reference patterns on which their performance is suboptimal by a factor of nearly d, for values of d, F and K that are typical in practice.

Theorem 1 *On any reference string R, the elapsed time of* conservative *with d disks on R is at most $d+1$ times the elapsed time of the optimal prefetching strategy on R.*

This bound is nearly tight for $d \ll F \ll K$: There are arbitrarily long strings on which conservative *requires time $1 + d\frac{K-F}{K}\frac{F}{F+d}$ times the optimal elapsed time.*

Theorem 2 *On any reference string R, the elapsed time of* aggressive *with d disks on R is at most $d(1 + \frac{F+1}{K})$ times the elapsed time of the optimal prefetching strategy on R.*

This bound is nearly tight for $d \ll \sqrt{F}$: There are arbitrarily long strings on which aggressive *requires time $d - \frac{3d(d-1)}{F+3(d-1)}$ times the optimal elapsed time (within an additive constant that depends only on F and K).*

Our main result is the development and analysis of a new algorithm, called *reverse aggressive*, whose performance is provably close to optimal. Interestingly, it achieves this by constructing a prefetching schedule backwards, i.e. by considering the reference sequence in reverse order. For reasons that will be made clear, this causes it to avoid problems encountered by the (forward) *aggressive* algorithm. *Aggressive* suffers from load imbalance and an inability to keep lightly loaded disks from outpacing (prefetching far ahead of) heavily loaded disks. We give an intuitive explanation as well as a more formal justification of *reverse aggresive*'s advantages in section 2. Detailed proofs are contained in the appendix. On real systems, dF/K is small[2], so that the factor $1 + dF/K$ in the following theorem is not much greater than one (hence our claim of "near-optimality").

Theorem 3 Reverse aggressive *requires at most $1 + dF/K$ times the optimal elapsed time to service any request sequence, plus an additive term dF independent of the length of the sequence.*

This bound is nearly tight for small d: There are arbitrarily long strings on which reverse aggressive *requires $(1 + (F-1)/K)$ times the elapsed time of the optimal prefetching strategy on R.*

1.4 Related Work

Our problem is a generalization of, but significantly more complicated than, the classical paging problem. Indeed, one principle for prefetching (the *optimal eviction* rule described in section 2.2) is derived from Belady's optimal *longest forward distance* [1] paging algorithm. As we will see, however, the application of this rule alone is insufficient to guarantee good prefetching performance; the natural algorithm based on it is suboptimal by a factor of nearly $d+1$. (See Theorem 1.)

On the theoretical side, we know of no prior work on the integration of parallel prefetching and caching. There have been some interesting results on the use of data compression for the design of optimal prefetching strategies [29, 49], and work on prefetching strategies for external merging under a probabilistic model of request sequences [38]. However, these studies concentrated only on the problem of determining which blocks to fetch, and did not address the problem of determining which blocks to replace.

Our work builds on recent studies of the sequential version of this problem (single disk) which showed [8, 7] that it is important to integrate prefetching, caching and disk scheduling and that a properly integrated strategy can perform much better than a naive strategy, both theoretically and in practice.

In the systems community, caching and prefetching have been known techniques to improve the performance of storage hierarchies for many years [50, 1, 17]. The breadth of application of these techniques has ranged from architecture [46] to database systems [47, 11, 39, 13] to file systems [17, 33, 24, 37, 48, 6, 21, 9, 42] and beyond. A recent trend in this research is to use applications' knowledge about their access patterns to perform more effective caching and prefetching [6, 9, 41, 42, 23, 34].

Our practical motivation for this problem comes from file systems. In this domain, the most common prefetching approach is to perform sequential read-ahead, i.e. to detect when an application accesses a file sequentially, and to prefetch the blocks of the files that are

[2] F/K is typically less than 0.02, and typical disk-arrays have at most 5 disks. Moreover, technological trends are such that F/K will only get smaller with time.

so used [17, 33, 35]. The obvious limitation of this approach is that it benefits only applications that make sequential references to large files. Another large body of work has been on predicting future access patterns (without hints from the application) even when access patterns are more complicated [16, 48, 39, 13, 21].

Much research in the past on parallel I/O has concentrated on techniques for "striping" and distributing error-correction codes among redundant disk arrays or other devices to achieve high bandwidth by exploiting parallelism and to tolerate failures [27, 45, 2, 12, 10, 40, 20, 31, 36, 19, 3, 5, 26, 43, 25, 14, 4, 18, 15, 22, 44].

Our work complements these previous efforts. File access prediction (with or without application hints) can be used to provide the inputs to the algorithm described in this paper. Once future accesses are known, our algorithm determines a near-optimal prefetching schedule. Our algorithm achieves near-optimal performance for any given layout of disk blocks (such as striping). Its performance will only improve when a near-optimal layout is used.

Recently, caching and prefetching have also been empirically studied for parallel file systems [16, 30, 32, 41, 51, 42].

Finally, in joint work with P. Cao, E. Felten and K. Li of Princeton University, we have performed an empirical study of the performance of the algorithms described in this paper. A companion paper [28] reports on this empirical evaluation. A brief summary of the results is given in section 3.4 of this paper.

1.5 Organization of the Paper

In section 2, we describe several properties that can be assumed of optimal prefetching algorithms. These constrain the problem and by adhering to them, we can ensure that an algorithm's performance is not far from optimal. Also in section 2, we describe the algorithms in greater detail and give intuition on their performance. In section 3, we give high level descriptions of the proofs of the results claimed in section 1.3. We conclude with open problems for further research. Detailed proofs are contained in the appendix.

2 Characterizing the Optimal Prefetching Schedule

2.1 Holes

At any point in processing the sequence (i.e. for any given cache state and cursor position), a *hole* is a block that is not present in the cache. We will use the term "hole" to refer to both the missing block and its next occurrence in the request sequence; which of these is meant will be clear from the context. If the cache is full, there are K out of $|B|$ blocks in the cache and thus $|B| - K$ holes. After a block is requested for the last time, we consider the corresponding hole in the request sequence to be at position $|R| + 1$, i.e., greater than the index of any request, where R is the request sequence.

2.2 Prefetching with a single disk

Before proceeding, we review the results of Cao et al. [8] for prefetching and caching in the single-disk case. They described four properties that can be assumed of any optimal strategy in the single-disk case:

1. *optimal fetching*: when fetching, always fetch the missing block that will be referenced soonest;
2. *optimal eviction*: when fetching, always evict the block in the cache whose next reference is furthest in the future;
3. *do no harm*: never evict block A to fetch block B when A's next reference is before B's next reference;
4. *first opportunity*: never evict A to fetch B when the same thing could have been done one time unit earlier.

It is easy to show that any schedule for serving requests and performing fetch-and-evict operations that does not follow these rules can be transformed into one that does, with performance at least as good. The first two rules specify what to fetch and what to evict, once a decision to fetch has been made. The last two rules constrain the times at which a fetch can be initiated. However, these rules do not uniquely determine a prefetching schedule. In particular, they do not specify how to choose between an earlier prefetch with a correspondingly earlier eviction and a later prefetch with a correspondingly later eviction. The former helps prevent stalling on earlier holes, whereas the latter may help prevent the introduction of holes, and hence stalling at a later time.

Nonetheless, these rules do provide a fair amount of guidance in the design of a prefetching algorithm. Cao et al. considered two natural algorithms that follow these rules, *aggressive* and *conservative*, that lie at opposite ends of the spectrum of possibilities. *Aggressive* is the algorithm that initiates a prefetch whenever its disk is ready (i.e. is not in the middle of a prefetch) and the *do no harm* rule allows it. *Conservative* is the algorithm that refuses to fetch until it can evict the same block that would be evicted by the optimal

longest forward distance [1] algorithm in the classical paging model. That is, *conservative* applies the rule *optimal eviction* as though the prefetch were to be initiated immediately before serving the request to the missing block, then applies the rule *first opportunity* to swap the chosen fetch/eviction pair as early as possible. *Conservative* makes the minimum number of total fetches, but it often declines opportunities to prefetch blocks.

Cao et al. showed that in the single-disk case, *conservative*'s elapsed time on any sequence is at most twice the optimal time, and that *aggressive*'s worst-case elapsed time is at most $\min(1+F/K,\ 2)$ times optimal, where F is the time required to fetch a block and K is the cache size measured in blocks. (They also showed that these bounds are tight.) On real systems, F/K is typically small, so *aggressive* is close to optimal.

2.3 The multi-disk case

There is an obvious and natural extension of each of these algorithms to the multi-disk case. For *aggressive*, it is the following: Whenever a disk is free, prefetch the first missing block of that disk's color, replacing the block (of any color) whose next reference is furthest in the future among all cached blocks. However, a fetch should be started only if the next access to the evicted block is after that to the block being fetched.

Unfortunately, as we shall see, this algorithm does not enjoy the same performance guarantee in the multi-disk case as it achieved in the single disk case. In fact, the four properties on which it was based in the single disk case do not hold for optimal strategies in the multi-disk case. As a result, it suffers from two problems in the multi-disk case that did not exist in the single disk case:

- The eviction decisions it makes are "color-blind": It chooses evictions to make without consideration of the load on the disks. These choices can result in a situation where many of the holes at any time are of the same color, and therefore can not subsequently be prefetched in parallel. (See figure 1 for an example of this.)
- *Aggressive* is too aggressive. The result is that it can cause some disks to fetch too far ahead with respect to other disks. These fetches increase the share of the cache occupied by blocks belonging to the lightly loaded disk(s), creating even more holes for the heavily loaded disk(s) to fill.

Therefore, we are motivated to approach the multi-disk prefetching problem in a way that will constrain the space of possibilities for the prefetching schedule in the same way that the four rules described above constrain the schedule in the single-disk case.

2.4 Properties of Optimal Parallel Prefetching

It is not hard to show that out of the four rules for optimal prefetching with one disk, only the last (*first opportunity*) holds when there are multiple disks. Finding a rule to replace *optimal fetching* is not much of a problem, however. The "colored" version of the rule can be used, i.e., for each disk c, the next block to fetch from c is the next missing block in the sequence that is colored c. Thus, as in the single-disk case, the question of which block to fetch reduces to the question of when to initiate a prefetch operation; this question needs to be answered for each disk, of course.

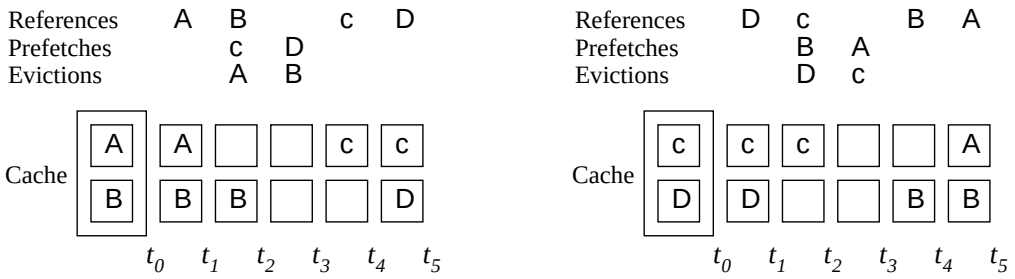
Optimal eviction is more troublesome. Suppose there are two disks, colored red and blue. If there are many red blocks missing in the sequence, say, it may be that the best choice for eviction is a blue block even though the block whose next request is furthest in the future is red. This is because the relatively lightly-loaded blue disk can better handle the increased burden of another missing block than the red disk can. Given that a blue block is to be evicted, say, it is true that the best choice is the blue block that is not requested for the longest time. That is, the colored version of this rule holds, but it doesn't tell us which color block to evict.

Even the seemingly obvious *do no harm* rule can be violated by the optimal prefetching strategy. This is because the loads on the disks can be imbalanced. If there are many red blocks missing from the sequence, say, but no blue blocks missing, it may be advantageous to buy time by evicting a blue block (and completing a fetch of a red block sooner than it would be possible otherwise), and then bringing the blue block back into the cache after a request to some red block has been served (so that a new eviction opportunity has arisen).

2.5 Using the reverse sequence

An interesting twist allows us to convert multiple-disk prefetching to a more constrained, and hence easier to solve, problem. In particular, we consider the request sequence in reverse (in a sense we will describe momentarily). We will be able to show that of the four rules, all but one (*optimal eviction*) hold for optimal schedules serving the reverse sequence. Moreover, we will be able to replace this rule by a simple "colored" variant (as we did with the *optimal fetching* rule for the forward sequence).

First, we return to the single disk case, and observe that any prefetching schedule that serves the reverse sequence S^r in time T can be used to derive a schedule to serve S in time T as follows. If the schedule for serving S^r serves request r_i between times t and $t+1$, the



(a) A forward schedule.

(b) A reversed schedule.

Figure 2: An example of reversing a schedule of prefetching and caching with two disks: a disk holding blocks A, B, and D, and another disk holding block c. The cache size is $K = 2$ and the fetch time is $F = 2$.

derived schedule for S serves r_i between times $T - t - 1$ and $T - t$. If the reverse schedule replaces a with b between times t and $t+F$, the derived schedule replaces b with a between times $T - t - F$ and $T - t$.[3] Applying this logic twice, we see that the optimal elapsed time for the reverse sequence is the same as the optimal elapsed time for the original sequence.

Reversal of the sequence is more complicated when multiple disks are considered. In the forward direction, the prefetching schedule is constrained to fetch at most one block at a time from each disk; eviction choices may be blocks of either color. Switching between the forward sequence and the reverse sequence, fetches become evictions and vice versa. To derive a useful schedule from a schedule serving the reverse sequence, then, requires that the schedule for the reverse sequence be constrained to *evict* at most one block of each color at a time. This is illustrated in the following example (see figure 2):

Consider the request sequence "ABcD", where upper case letters denote red blocks and lower case letters denote blue blocks. Let $F = 2$ and $K = 2$. By assumption, at time 0, blocks A and B reside in the cache (for the execution of the sequence in the forward direction). At time 1, a fetch is initiated to bring c into the cache from the blue disk, evicting A. At time 2, a fetch of D from the red disk is initiated, evicting B from the cache. The request to c stalls for one step until time 3 at which the fetch completes.

In the schedule for the reverse sequence, at time 1, D is evicted in order to start fetching B. Since c is blue and D is red, a fetch of A (evicting c) can be started at time 2, even though A and B are both red. The request for B stalls until time 3; it is served between times 3 and 4.

[3] We assume that all algorithms start with the cache containing the first K distinct requests in the sequence. Alternatively, all our results hold within an additive constant that accounts for differences in algorithms' transient cold-cache startup behaviors. We can assume without loss of generality that all algorithms end with the last K distinct requests in the cache.

As previously mentioned, all of the rules presented in section 2.2 except *optimal eviction* can be assumed of optimal prefetching schedules for the reverse sequence. *This fact makes it easier to find a schedule for the reverse sequence, then transform it into one for the original sequence, than to find a schedule for the original sequence directly.* The reason for this is that in the forward direction, any time a block is prefetched a decision must be made as to which color block to evict. In the reverse direction, this decision is made for us: the block to evict is the one not needed for the longest time whose color matches the color of the free disk. (I.e., the "colored" version of the *optimal eviction* rule can be used.) One might expect that fetch decisions are harder, but this is not the case. In the forward direction, the missing block to fetch is the one of the right color that is needed soonest. (This is the colored version of *optimal fetching* described earlier.) In the reverse direction, it is the one needed soonest, regardless of color.

2.6 The *reverse aggressive* algorithm

Reverse aggressive is a prefetching algorithm that performs aggressive prefetching on the reverse of its input sequence, then derives a schedule to serve the forward sequence as described in section 2.5. That is, on the reverse sequence, it behaves as follows. Whenever a disk is not in the middle of a prefetch, it determines which block in the cache is not needed for the longest time among those with the same color as the disk. If the index of the next request to that block is greater than the index of the first hole (of any color), a prefetch is initiated.

An intuitive explanation of *reverse aggressive*'s advantage over (forward) *aggressive* is the following:

- Whereas *aggressive* chooses evictions without considering the relative loads on the disks, *reverse aggressive* greedily evicts to as many disks as possible on the reverse sequence. In the forward direction,

this translates to making eviction decisions (creating new holes) based on the ability to fill the holes later on without incurring unnecessary stall penalties, while at the same time ensuring close to the highest degree of parallelism possible.

- Whereas *aggressive* can wastefully prefetch ahead on some of its disks, *reverse aggressive* is greedy in the reverse direction. Consequently, it is fetching pages in the forward direction just in time (to the extent possible) for them to be used. This results in performing close to the best evictions possible for those fetches, and exploiting parallelism as much as possible without creating load imbalance.

3 Results

In this section we give high-level descriptions of the main ideas used to derive our results. Full details are given in the appendix.

3.1 Performance of *conservative* and *aggressive*

The key concept in the proof of theorem 2 is the notion of *domination* from the work on prefetching in the single-disk case [8]. This allows us to bound the cost of *aggressive*'s prefetching schedule in terms of the progress of the optimal schedule at intermediate points during the processing of the request sequence.

Definition: Given two sets A and B of holes with $|A| \leq |B|$, A is said to *dominate* B if for all i, $1 \leq i \leq |A|$, the index of A's i^{th} hole (ordered by increasing index) is no less than the index of B's i^{th} hole. We will say that the i^{th} hole in A is *matched* to the i^{th} hole of B.

If *aggressive*'s cursor is ahead of *opt*'s cursor, and *aggressive*'s holes dominate *opt*'s holes, then *opt*'s cursor cannot pass *aggressive*'s: while *aggressive* stalls on a hole, *opt* cannot pass its matching hole. We show that *aggressive* is able to continually regain and maintain such an advantage (having its cursor ahead and its holes dominate) over *opt* at regular intervals, without losing too much time to *opt* in the process.

The lower bounds of nearly d in theorems 1 and 2 come from the fact that an adversary can construct request sequences that cause both *conservative* and *aggressive* to always fetch blocks from only one disk (because they make poor replacement choices). The optimal algorithm *opt* can serve these same sequences at nearly d times the rate because of the parallelism of prefetching on d disks. The additive term of one for *conservative* (in both the upper and lower bounds) comes from *opt*'s ability to overlap prefetches with the serving of requests. In contrast, *conservative* may not be able to do so.

The factor of d in the upper bounds comes from the fact that d is also a limit to the parallelism available to *opt*. As in the single-disk case, the factor of $1+(F+1)/K$ in the upper bound for *aggressive* comes from the fact that *aggressive*'s newly created holes are always at least K steps from the cursor. From this, it follows that *aggressive* prefetches too soon (creating extra holes) at most once every K requests.

3.2 Performance of *reverse aggressive*

The proof of theorem 3 required several new ideas. The notion of domination from the proof of theorem 2 was replaced by a stronger notion that we call *strong domination*.

Definition: Let A and B be sets of holes, possibly with different numbers of holes of each color. For each color c, let $N_c(A)$ (resp. $N_c(B)$) be the number of holes of color c in A (resp. B). Let $N_c = \min(N_c(A), N_c(B))$. If $N_c(A) > N_c(B)$, we say that c is an *excess color* of A; if $N_c(A) < N_c(B)$, c is an excess color of B; if $N_c(A) = N_c(B)$, c is not an excess color. Let $E_c = |N_c(A) - N_c(B)|$. If c is an excess color of A, we refer to A's first E_c holes of color c following the cursor as *excess* holes; excess holes of B are defined similarly. We say the set of holes A *strongly dominates* the set of holes B if

- for each c, A's last N_c holes of color c dominate B's last N_c holes of color c (i.e. A's non-excess holes of color c dominate B's non-excess holes of color c, whether c is an excess color of A or B or c is not an excess color), and
- all of B's excess holes precede the first hole in A of any color.

The following crucial lemma is then used to show that if *reverse aggressive* strongly dominates *opt*, and both have the opportunity to initiate a fetch replacing blocks of the same color, then *reverse aggressive* strongly dominates *opt* after the corresponding fetches complete.[4]

Lemma 4 Strong Domination Lemma

Let a and b be two prefetching schedules for a request sequence R that satisfy the optimal prefetching rules described in section 2.5. If a's holes at some time t

[4] We are speaking here of the performance of *reverse aggressive* on the reverse sequence, compared to an optimal schedule for the reverse sequence. However, as described in 2.5, the optimal elapsed time is the same in both directions, and from *reverse aggressive*'s schedule, we are able to derive a prefetching schedule for the forward sequence with the same elapsed time.

strongly dominate b's holes at some time t', a's cursor position at time t is at least as great as b's cursor position at time t', and

1. *both algorithms initiate a prefetch using the same disk (i.e. a and b evict blocks of the same color, a at time t and b at time t'), or*

2. *a initiates a prefetch at time t but b does not initiate a prefetch evicting a block of the same color at time t', or*

3. *b initiates a prefetch at time t' but a does not initiate a prefetch evicting a block of the same color at time t, and every block in a's cache of the same color as b's evicted block is requested before a's first hole at time t,*

then a's resulting holes strongly dominate b's resulting holes.

It is not possible to show that *reverse aggressive* strongly dominates *opt* throughout the sequence. Instead, we show that by giving *reverse aggressive* a little more time to serve every subsequence of K requests, it will strongly dominate *opt* at these regular intervals. That is, *reverse aggressive* loses about dF steps by prefetching too soon, thereby generating extra holes to fill, only every K requests or so.

The difficulty in showing this is that, in fact, *reverse aggressive* may prefetch prematurely very often, *but with at most $d-1$ disks.* We show that it is able to compensate by consistently making good (distant from the cursor) evictions with the other ("good") disk. While *reverse aggressive* spends an extra F steps relative to *opt* filling the first extra hole created by one of the "bad" disks, the good disk fills one hole. This gives *reverse aggressive* a "one hole lead" over *opt* with respect to the filling of holes. (Remember, each disk can fetch blocks of any color.) This provides a buffer against stalling on the (further) extra holes created by the bad disks, at least until an extra hole created by the good disk is reached. (The strong domination lemma is used to show that this invariant is maintained.) The good disk creates extra holes only once every K requests.

Formalizing all these arguments is difficult; the details are presented in the appendix.

3.3 The algorithms' running times

We mention only briefly the time required to determine a prefetching schedule (as opposed to the time required to serve the sequence in the model described in section 1.3, which is the primary measure we are trying to optimize). A linear pre-processing step can be used to determine per-block lists of requests (indices in the request sequence), per-disk lists of requests, per-disk lists of missing blocks (holes), and for each disk, the index of the next request to the block in the cache that is not needed for the longest time. These data structures can be maintained using a constant number of operations per request, so that each of the algorithms *conservative*, *aggressive*, and *reverse aggressive* can be implemented to run in time $O(|B| + |R|)$ in the uniform-cost RAM model. We omit the details of this analysis in this extended abstract.

3.4 Empirical Results

As mentioned, in joint work with P. Cao, E. Felten and K. Li, we have performed an empirical study of the performance of these algorithms. We implemented the *aggressive* and *reverse aggressive* algorithms and tested them on reference streams taken from real file systems. These results are reported in a companion paper [28].

We found that in practice, *aggressive* does substantially better than the worst-case performance we show here, if the layout of data on the disks is favorable (roughly, if the loads on the disks are balanced), though still not as well as *reverse aggressive*. With unbalanced loads on the disks (with the number of disks d ranging from 2 to 20), we found that *reverse aggressive* outperforms *aggressive* significantly. That paper also presents empirical results on the performance of limited-lookahead versions of these algorithms. In our simulations, we found that the algorithms perform well even with limited advance knowledge of future file accesses. For further details, see [28].

4 Open problems

As mentioned in the introduction, we know of no polynomial-time algorithm for optimal prefetching even for one disk. It is a difficult problem to find either such an algorithm or a proof of hardness. We do have an efficient algorithm to determine whether a sequence can be served with zero stall time (in the single-disk case).

Another very interesting direction is to extend these results to the case in which only probabilistic information is available about the request sequence.

Acknowledgements

We thank Richard Anderson for helpful discussions of this work and for commenting on a draft of this paper.

Pei Cao, Ed Felten and Kai Li introduced us to this problem, helped us to formalize the model and collab-

orated with us on the empirical evaluation. Their contribution to this work has been enormous.

Last, but not least, Martin Tompa's unfailing patience and encouragement have been indispensible to the first author. Both authors are extremely grateful to Martin for lending his insightful suggestions and constructive criticism to this work at every step of the way.

References

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

[2] D. Bitton and J. Gray. Disk Shadowing. In *Proceedings of the 14th VLDB Conference*, pages 331–338, Los Angeles, CA, August 1988.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 245–254, April 1994.

[4] L. Cabrera and D.D.E. Long. Swift: Using Distributed Disk Striping to Provide High I/O Data Rates. *Computer Systems*, 4(4):405–436, 1991.

[5] P. Cao, S.B. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 52–63, May 1993.

[6] Pei Cao, Edward Felten, and Kai Li. Application-Controlled File Caching Policies. In *USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[7] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. Technical Report TR-CS95-493, Princeton University, 1995.

[8] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of Integrated Prefetching and Caching Strategies. In *Proceedings of the ACM SIGMETRICS*, May 1995.

[9] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–178, November 1994.

[10] P.M. Chen and D.A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 322–331, May 1990.

[11] H.T. Chou and D.J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 127–141, Dublin, Ireland, 1993.

[12] B.E. Clark and et al. Parity Spreading to Enhance Storage Access. United States Patent 4,761,785, August 1988.

[13] Kenneth M. Curewitz, P. Krishnan, and Jeffrey S. Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, pages 257–266, Washington, DC, May 1993.

[14] P.C. Dibble, M.L. Scott, and C.S. Ellis. Bridge: A High-Performance File System for Parallel Processors. In *the 8th International Conference on Distributed Computing*, pages 154–161, 1988.

[15] A.L. Drapeau, K.W. Shirriff, J.H. Hartman, E.L. Miller, S. Seshan, R.H. Katz, K. Lutz, and D.A. Patterson. RAID-II: A High-Bandwidth Network File Server. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 234–244, April 1994.

[16] Carla Schlatter Ellis and David Kotz. Prefetching in File System for MIMD Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 306–314, August 1989.

[17] R.J. Feiertag and E.I. Organisk. The Multics Input/Ouput System. In *Proceedings of the 3rd Symposium on Operating Systems Principles*, pages 35–41, 1971.

[18] D.G. Feitelson, P.F. Corbett, J.P. Prost, and S.J. Baylor. Parallel Access to Files in the Vesta File System. In *Supercomputing '93*, pages 472–483, November 1993.

[19] G.A. Gibson. Redundant Disk Arrays: Reliable, Parallel Secondary Storage. The MIT Press, Cambridge, Massachusetts, 1992.

[20] James Gray and et al. Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput. In *16th International Conference on VLDB*, Australia, 1990.

[21] Jim Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[22] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[23] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[24] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[25] Intel Supercomputer Systems Division, In Chapter 5, *Paragon User Guide. Using Parallel File I/O*, November 1993.

[26] R. H. Katz, P. M. Chen, A. L. Drapeau, E. K. Lee, E. L. Miller, S. Seshan, and D. A. Patterson. RAID-II: Design and Implementation of a Large Scale Disk Array Controller. In *VLSI System Design Conference*, Seattle, WA, March 1993.

[27] M. Kim. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, 35(11):978–988, 1986.

[28] Tracy Kimbrel, Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Integrated Parallel Prefetching and Caching. Short version to appear in *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

[29] P. Krishnan and Jeffrey S. Vitter. Optimal Prediction for Prefetching in the Worst Case. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1994.

[30] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[31] E. Lee and R. Katz. Performance Consequences of Parity Placement in Disk Arrays. In *Proceedings of The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–199, 1991.

[32] K-K. Lee and P. Varman. Prefetching and I/O Parallelism in Multiple Disk Systems. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995.

[33] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[34] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the First USENIX Mach Symposium*, 1990.

[35] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, 1991.

[36] Jai Menon and Dick Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 318–329, May 1992.

[37] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[38] Vinay S. Pai, Alejandro A. Schäffer, and Peter J. Varman. Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging. *Theoretical Computer Science*, v. 128, 1994.

[39] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September 1991.

[40] D.A. Patterson, G. Gibson, and R.H. Katz. A Case for Redundant Arrays for Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD Conference*, pages 109–116, June 1988.

[41] R. Hugo Patterson and Garth A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

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

[43] Paul Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of 4th Conference on Hypercubes*, March 1989.

[44] Christos A. Polyzois, Anupaam Bhide, and Daniel M. Dias. Disk Mirroring with Alternating Deferred Updates. In *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, 1993.

[45] K. Salem and H. Garcia-Molina. Disk Striping. In *the 2nd IEEE Conference on Data Engineering*, 1986.

[46] Alan J. Smith. Second Bibliography on Cache Memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[47] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[48] C. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *Proceedings of Parallel and Distributed Information Systems*, pages 190–196. IEEE, 1991.

[49] Jeffrey S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 121-130, 1991.

[50] Maurice Wilkes. Slave Memories and Dynamic Storage Allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[51] Kun-Lung Wu, Philip S. Yu, and James Z. Teng. Performance Comparison of Thrashing Control Policies for Concurrent Mergesorts with Parallel Prefetching. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 171–182, 1993.

Appendix: Proofs

Terminology

The following definitions will be useful. Further definitions will be introduced later, which will be specific to the particular proofs in which they are used.

Definition: We divide the request sequence into *phases*, maximal-length subsequences of requests to K distinct blocks, as follows. The first phase begins with the first request. Each phase ends immediately before the first request to the $(K+1)^{st}$ distinct block since the beginning of the phase, and the next phase begins with that request.

Given two sets A and B of holes with $|A| \leq |B|$, A is said to *dominate* B if for all i, $1 \leq i \leq |A|$, the index of A's i^{th} hole (ordered by increasing index) is no less than the index of B's i^{th} hole. We will say that the i^{th} hole in A is *matched* to the i^{th} hole of B. Notice that domination is transitive.

If algorithm A has fetches in progress at any time t, we denote A's holes before initiating those fetches by $H_A^-(t)$ (i.e. H_A^- includes the holes being filled, but not the ones being created), and A's holes after those fetches complete (but ignoring any later fetches that may overlap them) by $H_A^+(t)$.

Reverse aggressive: upper bound

Notation

- In this section, we assume all algorithms are working with the reverse sequence, and denote the optimal algorithm for serving the reverse sequence by *opt*.
- Under any algorithm that works on the forward sequence and follows the *optimal eviction* rule, no new holes will be created in a phase once the cursor enters the phase. For every hole in the phase, there is at least one block in the cache that is not requested for the remainder of the phase (since there are only K blocks requested in the phase, by definition, and the cache holds K blocks). In contrast, it is possible that *reverse aggressive* (and *opt* working on the reverse sequence, in fact) will create a new hole within a phase even after its cursor has entered the phase. Although it's true that for every hole in the phase, there is a block in the cache that is not requested until after the end of the phase, it may be that all those blocks are the same color, and that the best eviction choice of another color is a block that will be requested before the end of the phase. However, if *reverse aggressive* does create new holes in the phase containing the cursor, it will create such holes of at most $d-1$ colors. We refer to the other disk as the *busy* disk for the phase. As long as there are holes remaining in the phase, the busy disk will initiate a fetch to fill one of them every F steps.
- A fetch using the busy disk (and evicting a block of the same color as the busy disk; the block fetched may be any color) is referred to as a *busy-disk fetch*; fetches using other disks are referred to as *non-busy-disk fetches*.

Outline of the proof

We first give some preliminaries, proving the claims of section 2.4 and a simple lemma on combining subsets of dominating and dominated sets of holes. Next, we define strong domination, the invariant that is maintained as we compare *reverse aggressive*'s prefetching schedule to *opt*'s, and a variant of this we call *phase domination*. We show that if *reverse aggressive* strongly dominates (or phase dominates) opt, and both initiate a prefetch operation using the same disk (i.e. evicting blocks of the same color) according to the rules for optimal prefetching, *reverse aggressive* continues to strongly dominate (resp. phase dominate) *opt* after their respective prefetches are completed.

The phase domination lemma is used to bound *reverse aggressive*'s elapsed time for a single phase relative to *opt*'s elapsed time. Roughly speaking, if *reverse aggressive*'s holes dominate *opt*'s, *opt* can't get ahead of *reverse aggressive* since *opt*'s first hole is at least as early in the request sequence as *reverse aggressive*'s. By allowing *reverse aggressive* a small amount of time to correct for mistakes it makes by prefetching sooner than *opt*, the phase domination invariant is maintained until both algorithms reach the end of the phase. This step of the proof is complicated by the fact that the algorithms may fetch blocks using their respective disks in different orders. We must permute one sequence of fetches in order to make direct comparisons between the two algorithms' operations.

Finally, we show that by using a different permutation (and a correspondingly different matching of one algorithm's prefetch operations to the other's), the strong domination lemma implies that strong domination holds as an invariant as we compare the algorithms' progress from one phase to the next.

Lemma 5 *Any prefetching schedule that doesn't satisfy the four rules described in section 2.4 can be transformed into one that does, with no increase in elapsed time.*

Proof:

1. *optimal fetching* (fill the first hole): Suppose that at time t_1, a fetch is initiated to fill some hole h_2 other than the first hole h_1. h_1 must be filled before it can be served; say it is filled by a fetch initiated at time $t_2 > t_1$. Since the (later) reference to h_2 cannot be served until after the reference to h_1 is served, no further stall time is induced by filling h_1 at time t_1 and h_2 at time t_2. Since we are working

with the reverse sequence, this change can be made regardless of the colors of h_1 and h_2.

2. *colored optimal eviction* (evict the block not needed for the longest time among those colored the same as the free disk): Suppose that at time t_1, block b_1 is evicted, and block b_2 of the same color as b_1 is in the cache and is not referenced before the next reference to b_1. If b_2 is subsequently evicted before the next reference to b_1 is served, the effect is the same if b_2 is evicted first, then b_1. Otherwise, b_1 must be fetched back at some time $t_2 > t_1$ before the reference to it can be served. If b_2 is evicted at time t_1 instead of b_1, it can be fetched back at time t_2. By assumption, there are no intervening references of b_2 on which to stall; thus the transformed schedule stalls no more than the original.

3. *do no harm* (don't evict b_1 to fetch b_2 if b_1 is needed sooner): Suppose b_1 is evicted to fetch b_2. b_1 must be fetched back before the reference to it can be served; this fetch evicts some other block b_3. Since fetches on any disk can be of any color, the fetch of b_1 can be replaced by a fetch of b_2 (evicting b_3). By assumption, there are no intervening references of b_2 on which to stall; thus the transformed schedule stalls no more than the original.

4. *first opportunity* (perform each fetch/eviction pair as soon as possible): Suppose that disk c is left idle at time t, a fetch of block b_1 is initiated at $t+1$ evicting block b_2 of color c, and that the block served at time t is not b_2. Then by initiating the fetch at time t rather than $t+1$, the hole (b_1) is filled one step sooner; certainly, no additional stall is incurred by this change.

□

Lemma 6 *Given two sets of holes $A = A_1 \cup A_2$ and $B = B_1 \cup B_2$ with $|A_1| \leq |B_1|$, $|A_2| \leq |B_2|$, $A_1 \cap A_2 = \emptyset$, and $B_1 \cap B_2 = \emptyset$, if A_1 dominates B_1 and A_2 dominates B_2, then A dominates B.*

Proof: Suppose the contrary. Let i be such that the i^{th} member of A (ordered, as usual, by increasing index in the request sequence) has an index less than the i^{th} member of B. Then A contains i holes with indices less than or equal to that of A's i^{th} hole, and B contains only $i-1$ such holes. But because A_1 dominates B_1 and A_2 dominates B_2, for each member of A there is a distinct member of B with lesser or equal index. Thus we have a contradiction. □

Note that the lemma extends to pairs of sets composed of more than two disjoint subsets each.

Definition: Let A and B be sets of holes, possibly with different numbers of holes of each color. For each color c, let $N_c(A)$ (resp. $N_c(B)$) be the number of holes of color c in A (resp. B). Let $N_c = \min(N_c(A), N_c(B))$. If $N_c(A) > N_c(B)$, we say that c is an *excess color* of A; if $N_c(A) < N_c(B))$, c is an excess color of B; if $N_c(A) = N_c(B)$, c is not an excess color. Let $E_c = |N_c(A) - N_c(B)|$. If c is an excess color of A, we refer to A's first (counting from the cursor to the end of the sequence) E_c holes of color c as *excess* holes; excess holes of B are defined similarly.

We say the set of holes A *strongly dominates* the set of holes B if

- for each c, A's last N_c holes of color c dominate B's last N_c holes of color c (i.e. A's non-excess holes dominate B's non-excess holes, whether c is an excess color of A or B or c is not an excess color), and
- all of B's excess holes strictly precede the first hole in A of any color.

This idea is illustrated in figure 3, in which holes of different colors are depicted by different shapes.

Notice that by the previous lemma, strong domination implies (ordinary, color-blind) domination. (Match non-excess holes according to colors, and all of one set's excess holes to all of the other set's excess holes.) We will be concerned most of the time with equal-sized sets of holes. The exception to this is merely a convenience that allows us to consider separately the effects of changes to sets of holes, where the changes always occur in pairs that conserve the sizes of the sets (i.e. one hole is filled and a new one created by a prefetch operation).

Lemma 7 *Strong domination is transitive.*

Proof: Suppose A strongly dominates B and B strongly dominates C. We show that A strongly dominates C. Fix a color c; for convenience (so we can use it as an adjective), suppose c is red. Define $N_c(\cdot)$ as before. For a collection S of sets of holes, let $N_c(S) = \min_{s \in S}(N_c(s))$. (We will drop the brackets when listing the members of a set.) Let $N_c = N_c(A, B, C)$. We consider three cases:

1. $N_c = N_c(A)$. A has N_c red holes, and these dominate the last N_c red holes in B. B's last $N_c(B, C)$ red holes dominate C's last $N_c(B, C)$ red holes, so B's last N_c red holes must dominate C's last N_c red holes. Since domination is transitive, A's N_c red holes dominate C's last N_c red holes. Suppose h is a red hole in C that is excess with respect to A. If h is matched to a red hole h' of B, h' is excess w.r.t. A and thus precedes A's first hole, so h must precede A's first hole as well. If h is excess w.r.t. B, it precedes B's first hole, which precedes or is the same as A's first hole.

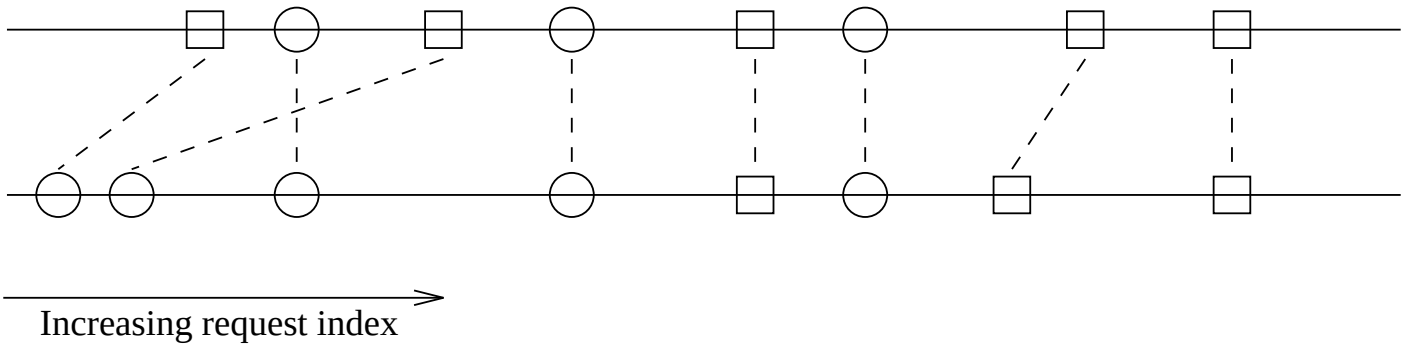


Figure 3: Strong domination example: the upper set of holes strongly dominates the lower one. Mismatched shapes represent excess holes.

2. $N_c = N_c(B)$. A's last N_c red holes dominate B's N_c red holes, which dominate C's last N_c red holes. Suppose h is a red hole in C that is excess w.r.t. B. h must precede B's first hole. Since B's first hole precedes or is the same as A's first hole, h precedes A's first hole as well. If h is excess w.r.t. A, we are done. If h matches some hole h' of A, h surely does not occur after h'.

3. $N_c = N_c(C)$. A's last $N_c(A, B)$ red holes dominate B's last $N_c(A, B)$ red holes, so A's last N_c red holes must dominate B's last N_c red holes, which dominate C's N_c red holes. C has no excess red holes w.r.t. B or A.

These three cases are illustrated in figure 4. □

Lemma 8 Domination Lemma

Let a and b be two prefetching schedules for a request sequence R. If a's holes at some time t strongly dominate b's holes at some time t', a's cursor position at time t is at least as great as b's cursor position at time t', and

1. *both algorithms perform a prefetch using the same disk (i.e. a and b evict blocks of the same color, a at time t and b at time t'), or*

2. *a performs a prefetch at time t but b does not at time t', or*

3. *b performs a prefetch at time t' but a does not at time t, and every block in a's cache of the same color as b's evicted block is requested before a's first hole (i.e. a prefetches aggressively),*

then a's resulting holes strongly dominate b's resulting holes.

This is illustrated in figure 5.

Proof: Define $N_c(A)$, $N_c(B)$, and N_c as before (with respect to the sets of holes *before* each change to a set of holes is made), where A denotes a's set of holes at time t and B denotes b's set of holes at time t'.

We consider the individual changes to A and B in three steps:

1. a's first hole is removed from A if one of clauses 1 and 2 of the premise holds.

2. b's new hole is added to B if one of clauses 1 and 3 holds and a's new hole is added to A if one of clauses 1 and 2 holds.

3. b's first hole is removed from B if one of clauses 1 and 3 holds.

We will show that after each step, strong domination of A over B is preserved.

For convenience, we will say that (a hole at) index i is "left" of (a hole at) index j, and (the hole at) j is "right" of (the hole at) i, if $i < j$.

First, suppose that clause 1 of the premise holds.

Step 1: a's first hole is filled

Let c be the hole's color. First, since a's new first hole is to the right of its old first hole (the one being filled), b's excess holes all are still to the left of a's first hole. If c was an excess color of a, we are done. Otherwise, b's hole that was matched to a's filled hole becomes an excess hole, and since it occurred no later than the hole it matched, it is to the left of a's new first hole.

Step 2: eviction

Let a's last N_c holes of the same color c as the block evicted occur at indices $a_1 < a_2 < \ldots < a_{N_c}$, and let b's occur at $b_1 < b_2 < \ldots < b_{N_c}$. Since a's holes strongly dominate b's, we know that $a_i \geq b_i$ for each i. Let a and b both make the best possible eviction of color c. Let b's new hole be its j^{th} non-excess hole of color c, i.e. the new hole occurs between b_{j-1} and b_j, or at an index greater than b_{N_c} in which case $j = N_c + 1$, or before b_1 in which case $j = 1$. (As a special case, if c is an excess

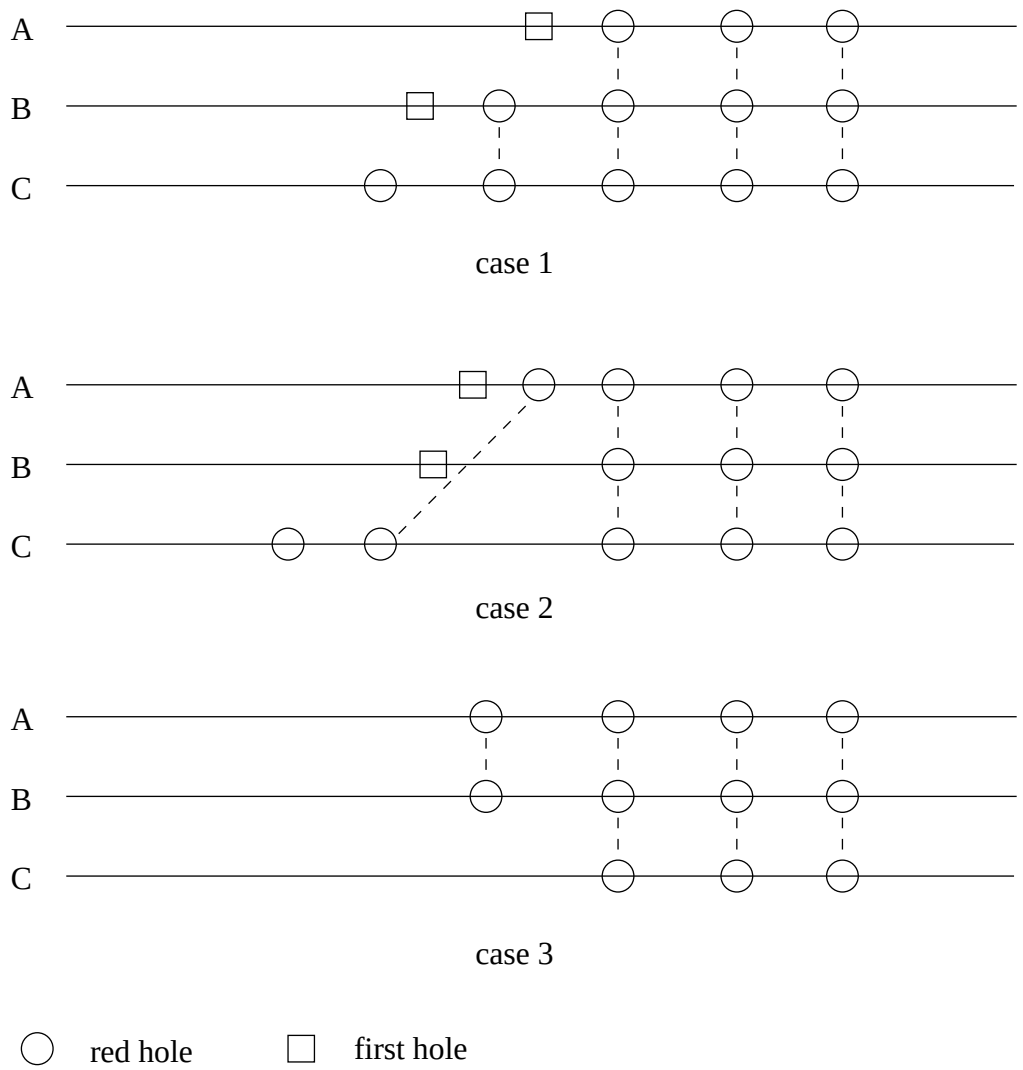


Figure 4: Strong domination is transitive.

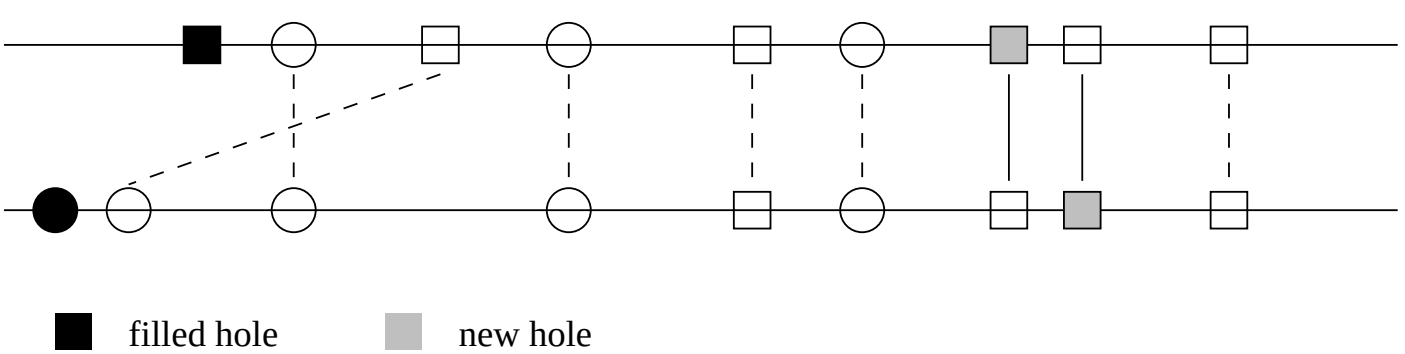


Figure 5: Domination lemma: the upper set of holes continues to strongly dominates the lower one.

color of b, and the new hole is before b's last excess hole of color c, the new hole becomes an excess hole and the last excess hole takes its place in the following argument.) Let a's new hole be its r^{th} hole of color c, with a similar special case to that in the definition of j. Let $a'_1 < a'_2 < \ldots < a'_{N_c+1}$ be the indices of a's last $N_c + 1$ holes of color c after the eviction, and let $b'_1 < b'_2 < \ldots < b'_{N_c+1}$ be the indices of b's last $N_c + 1$ holes of color c after the eviction. Then for $i < r$, $a'_i = a_i$ and for $i > r$, $a'_i = a_{i-1}$; for $i < j$, $b'_i = b_i$ and for $i > j$, $b'_i = b_{i-1}$. To show that domination is preserved, we need to show that $a'_i \geq b'_i$ for each i, $1 \leq i \leq N_c + 1$. For $i < \min(r, j)$ and $i > \max(r, j)$ it is immediate that $a'_i \geq b'_i$. If $r > j$, then we have

$$\begin{aligned} a'_r > a_{r-1} &\geq b_{r-1} = b'_r \\ a'_{r-1} = a_{r-1} &\geq b_{r-1} > b'_{r-1} \\ &\ldots \\ a'_j = a_j &\geq b_j > b'_j \end{aligned}$$

and we are done. If $r \leq j$, then we must show

$$\begin{aligned} a'_j = a_{j-1} &\geq b'_j \\ a'_{j-1} = a_{j-2} &\geq b'_{j-1} = b_{j-1} \\ &\ldots \\ a'_{r+1} = a_r &\geq b'_{r+1} = b_{r+1} \\ a'_r &\geq b'_r = b_r. \end{aligned}$$

Suppose that one or more of these inequalities does not hold, and let i be the largest index for which $a'_i < b'_i$. Then we have

$$a'_i < b'_i < b'_{i+1} \leq a'_{i+1}$$

where a's new hole at a'_r satisfies $a'_r \leq a'_i$. But this means that a had a block that is not requested until index b'_i in its cache, and elected to evict the block requested earlier at index a'_r instead. This contradicts the assumption that a made the best possible eviction choice, i.e. that it evicted the block whose next occurrence was at the greatest index among all blocks in the cache.

Since the holes of color other than c are unaffected by this change, and domination of holes of color c is preserved, strong domination is preserved.

Step 3: b's first hole is filled

Let c be the hole's color. If c is an excess color of b, then b will have one fewer excess hole of color c; the remaining ones are unchanged, and thus are still to the left of a's first hole. Otherwise, the hole was matched to some hole of a, which becomes an excess hole. The newly excess hole's position is relevant in the definition of strong domination only if it is a's first hole; in this case, since neither a's first hole nor b's excess holes are changed, strong domination is preserved.

The proof is complete for clause 1 of the premise.

If clause 2 holds (a fetches but b does not), step 1 is the same as in the proof for clause 1. For step 2, first note that a's new hole is to the right of a's (old) first hole (by the *do no harm* rule), so that b's excess holes still precede all of a's holes. Let c be the color of a's new hole. If c is an excess color of b, an argument similar to the one above for clause 1 shows that a's holes of color c will dominate b's non-excess holes of the same color. If c is not an excess color of b, the new hole or some previous hole of a will become an excess hole. In the former case, a's last N_c holes are unchanged. In the latter case, the index of a's i^{th} non-excess hole of color c is the same or greater than before, for each $i \leq N_c$. No changes are made in step 3.

If clause 3 holds (b fetches but a does not), nothing happens in step 1. Let c be the color of b's new hole. Again, for step 2, an argument similar to that for clause 1 shows that a's non-excess holes of color c dominate b's non-excess holes of color c; if not, a would perform a prefetch operation since a prefetches aggressively. If c is not an excess color of b, we are done with step 2. Otherwise, we need to show that all of b's excess holes of color c precede a's first hole. Suppose that b has $N_c + 1$ holes of color c at or to the right of a's first hole. a has only N_c holes of color c, so b has some hole h of color c that is not a hole of a and is to the right of a's first hole. Again, we have a contradiction to the hypothesis that a prefetches aggressively: a can fill its first hole and evict the block requested at h. Step 3 is the same as for clause 1. □

Phase domination is similar to strong domination, but is concerned only with holes in the phase containing the cursor.

Definition: Let A (B) be algorithm a's (resp. b's) set of holes at time t (resp. t') such that a's cursor at time t is in the same phase of the request sequence as b's cursor at time t'. For each color c, define $N_c(A)$, $N_c(B)$, N_c, E_c, and excess holes as before. Then A phase-dominates B if

1. All of b's excess holes strictly precede a's first hole of any color.
2. For each color c that is excess for a, if b's i^{th} hole of color c is within the phase for any $i > 0$, it occurs no earlier than a's $(i + E_c)^{th}$ blue hole.
3. For each color c that is not an excess color or is an excess color for b, if b's i^{th} hole of color c is within the phase for any $i > E_c$, it occurs no earlier than a's $(i - E_c)^{th}$ hole of color c.

Lemma 9 *Phase domination is transitive.*

Proof: Similar to lemma 7. □

The proof of the following is the same as that of of lemma 8, except we consider all holes beyond the end of the phase boundary to be equivalent (i.e. as though they were beyond the end of the entire request sequence).

Lemma 10 Phase Domination Lemma

Let a and b be two prefetching schedules for a request sequence R. If a's holes at some time t phase dominate b's holes at some time t',

1. *both algorithms perform a prefetch using the same disk (i.e. a and b evict blocks of the same color, a at time t and b at time t'), or*
2. *a performs a prefetch at time t but b does not at time t', or*
3. *b performs a prefetch at time t' but a does not at time t, and every block in a's cache of the same color as b's evicted block is requested before a's first hole,*

and a's cursor position at time t is at least as great as b's cursor position at time t' or *the blocks evicted are the same color as a's busy disk in the current phase (and thus the new holes are beyond the end of the current phase), then a's resulting holes phase dominate b's resulting holes.*

Here is our main result.

Theorem 11 Reverse aggressive *requires at most* $1 + dF/K$ *times the optimal elapsed time to service any request sequence, plus an additive term dF independent of the length of the sequence.*

Proof: We show that for each i, there are times T_i and T_i', such that

- *reverse aggressive*'s cursor at time T_i is not more than $F-1$ steps behind *opt*'s cursor at time T_i';
- *reverse aggressive*'s cursor is within the i^{th} phase;
- $H^+_{agg.}(T_i)$ dominates $H^-_{opt}(T_i')$;
- Neither *reverse aggressive* nor *opt* is in the middle of a fetch on *reverse aggressive*'s busy disk for phase i;
- $T_i' + i(dF - 1) \geq T_i$.

The theorem will follow from the last condition, since each phase is of length at least K, so that *opt*'s elapsed time is at least K for each phase.

We prove this by induction. For the base case ($i = 0$), we take $T_0 = T_0' = 0$. The fact that the claims hold at this time is trivial. For the inductive step, assume the claims hold for the i^{th} phase. We show that they hold for the $(i+1)$-st phase via a two step process.

- We first show that in phase i, *reverse aggressive* loses at most $(d-1)F$ steps to *opt* (lemma 12).
- We then use this fact to show that at the end of the phase, by giving *reverse aggressive* an extra $dF - 1$ steps relative to *opt* (from the start of the phase), the invariants are restored.

We begin with a formal statement of the first of these steps.

Lemma 12 *Suppose that at time T_i,* reverse aggressive*'s cursor is at position p_i in the sequence. Let $T_i' + t_O(j)$ (resp. $T_i + t_A(j)$) denote the time at which* opt *(resp.* reverse aggressive*) serves the request at cursor position $j \geq p_i$, for any j such that r_j is in phase i. Then for all j in the phase, $t_A(j) \leq t_O(j) + (d-1)F$.*

Proof: Suppose the contrary, and consider the least index ℓ such that $t_A(\ell) > t_O(\ell) + (d-1)F$.

Then $t_A(\ell - 1) \leq t_O(\ell - 1) + (d-1)F$, and *reverse aggressive* stalls at least one step more than *opt* on request r_ℓ. In particular, *reverse aggressive* stalls at time $T_i + t_A(\ell) - 1$, and *opt* does not stall at time $T_i' + t_O(\ell)$.

We know that *reverse aggressive* will perform busy-disk fetches continuously (completing a fetch at time $T_i + bF$ for each $b \geq 1$) at least until such a time as there are no holes left in the phase (after which *reverse aggressive* won't stall at least until the end of the phase is reached). Now, let b and δ be such that $t_A(\ell) - 1 = bF + \delta$ and $\delta < F$. Then *reverse aggressive* has filled b holes by busy-disk fetches by time $T_i + t_A(\ell) - 1$, and *opt* has filled at most $b - d + 1$ holes by busy-disk fetches by time $T_i' + t_O(\ell)$, since

$$\begin{aligned} t_O(\ell) &< t_A(\ell) - (d-1)F \\ &= bF + \delta + 1 - (d-1)F \\ &\leq (b - d + 2)F. \end{aligned}$$

Let n be the number of non-busy-disk fetches completed by *opt* by time $T_i' + t_O(\ell)$. Consider the sequence $S = ((c_1, color_1), \ldots, (c_{n+b-d+1}, color_{n+b-d+1}))$ of fetches *opt* initiates after time T_i' that complete at or before time $T_i' + t_O(\ell)$, where the pair $(c, color)$ denotes that a fetch evicting a block of color *color* is initiated at cursor position c. For each fetch $(c', color')$ of *opt*, we define a *matching fetch opportunity* of *reverse aggressive*. A matching fetch opportunity is a pair $(c, color)$ such that *reverse aggressive* has the opportunity to initiate a fetch of color *color* at a cursor position at least

as great as c. Each matching fetch opportunity to a fetch in S allows *reverse aggressive* to complete a fetch (if necessary) by time $T_i + t_A(\ell) - 1$. They are defined as follows:

- *opt*'s j^{th} busy-disk fetch is matched to the j^{th} busy-disk fetch *reverse aggressive* performs in the phase. (Since *reverse aggressive* prefetches continuously using its busy disk, we know that each of these fetch opportunities corresponds to an actual fetch.)
- Let *opt*'s j^{th} non-busy-disk fetch be initiated at time $T'_i + t'_j$. This fetch is matched to the fetch on the same disk that *reverse aggressive* initiates (if any) in the time interval

  $$[T_i + t'_j + (d-1)F, T_i + t'_j + dF - 1].$$

  Note that by hypothesis, at time $T_i + t'_j + (d-1)F$ *reverse aggressive* is already at or beyond the cursor position at which *opt* initiates its j^{th} non-busy-disk fetch, and its disk of the same color becomes free (finishes any fetch already in progress) within another $F-1$ steps. Therefore, such a fetch opportunity exists.

  If *opt* completes a total of n non-busy-disk fetches by time $T'_i + t_O(\ell)$, then each fetch except (possibly) the last one on each non-busy-disk (i.e. at least $n-(d-1)$ of the n non-busy-disk fetches) is initiated at a time less than or equal to $T'_i + t_O(\ell) - 2F$. Therefore, *reverse aggressive* can initiate a matching fetch if needed at a time strictly less than $T_i + t_O(\ell) + (d-2)F$ and will complete the fetch at a time strictly less than

  $$T_i + t_O(\ell) + (d-1)F < T_i + t_A(\ell).$$

- Finally, the last non-busy-disk fetch of each color performed by *opt* is matched to one of the last $d-1$ busy-disk fetches performed by *reverse aggressive*.

We claim that *reverse aggressive*'s holes after these $n+b-d+1$ matching fetch opportunities phase dominate *opt*'s holes after completing its sequence S of n non-busy-disk fetches and at most $b-d+1$ busy-disk fetches. Let A_0 be *reverse aggressive*'s set of holes at time T_i. Let O_0 be *opt*'s set of holes at time T'_i. Let $New(H, (c, color))$ denote the new set of holes (uniquely determined by the optimal prefetching principles *optimal fetching* and *colored optimal eviction* described in section 2.5) should a prefetch be initiated, if possible (i.e. if allowed by the *do no harm* principle), evicting a block of color *color* at cursor position c when the current set of holes is H. Define O_j, $j \geq 1$, inductively as the set of holes resulting from executing *opt*'s j^{th} fetch $(c_j, color_j)$ with the set of holes O_{j-1}; i.e. $O_j = New(O_{j-1}, (c_j, color_j))$. Similarly, define A_j, $j \geq 1$, inductively by $A_j = New(A_{j-1}, (c_j, color_j))$. $A_{n+b-d+1}$ is the state that would be reached by starting in *reverse aggressive*'s state A_0, but then initiating fetches (when possible) according to *opt*'s prefetching schedule. By a sequence of applications of the phase-domination lemma (Lemma 10), we have that $A_{n+b-d+1}$ phase-dominates $O_{n+b-d+1}$.

We now show that *reverse aggressive*'s holes after exercising its matching fetch opportunities phase dominate $A_{n+b-d+1}$. Because phase domination is transitive, we will obtain that *reverse aggressive*'s holes phase dominate *opt*'s. Since *opt* and *reverse aggressive* may perform fetches on different disks at different times and in different orders, we need to somehow permute *opt*'s schedule of fetches into *reverse aggressive*'s; then we will be able to make pairwise comparisons between the two sequences of fetches and apply the phase domination lemma. Toward this end, we define the following:

Definition: Consider a fetch sequence, defined by a sequence of triples of the form $(t_j, c_j, color_j)$, where for each j, $t_j \leq t_{j+1}$ and $c_j \leq c_{j+1}$. $(t_j, c_j, color_j)$ denotes a fetch, or an attempt at a fetch (since no fetch may be possible under the optimal prefetching rules), beginning at time t_j with the cursor at a position at least c_j, where the color of the disk performing the fetch (and the color of the evicted block) is $color_j$.

A fetch sequence S is obtained from a fetch sequence S' by a *busy-early swap* if S' and S are the same except that a pair $(t'_j, c'_j, color_j)$, $(t'_{j+1}, c'_{j+1}, color_{j+1})$ in S' is replaced by $(t_j, c_j, color_{j+1})$, $(t_{j+1}, c_{j+1}, color_j)$ in S, where $c_j \geq p_i$, $c_{j+1} \geq c'_j$, and $color_{j+1}$ is the color of *reverse aggressive*'s busy disk for the phase. ($c_j \geq p_i$ will be enough to ensure that *reverse aggressive* is able to complete a fetch with the busy disk and that the new hole is beyond the end of phase i, which is what is needed to maintain phase domination.)

A fetch sequence S is obtained from a fetch sequence S' by an *overlapping swap* if S and S' are the same except that a pair $(t'_j, c'_j, color_j)$, $(t'_{j+1}, c'_{j+1}, color_{j+1})$ in S' is replaced by $(t_j, c_j, color_{j+1})$, $(t_{j+1}, c_{j+1}, color_j)$ in S, where $t'_{j+1} < t'_j + F$, $t_{j+1} < t_j + F$, $c_j \geq c'_{j+1}$, and $c_{j+1} \geq c'_j$. (Note that for actual fetch sequences, $c_{j+1} \geq c'_j$ is implied by $c_j \geq c'_{j+1}$, since cursor positions increase with time.)

Lemma 13 *Suppose that fetch sequence S within phase i is obtained from fetch sequence S' by a busy-early swap. Then the set of holes reached by performing S phase dominates that reached under S'.*

Proof: Let blue denote the color of the busy disk, and let red denote the color of the second disk to fetch (under S') in the swapped pair. The sets of holes of the two sequences immediately before completing the swapped pair of fetches are the same. In both cases, a

blue fetch can be performed (since by hypothesis there are still holes in the phase), and will not create a new hole within the phase.

Unless the first hole is a red block, the set of red blocks in the cache at the time of the red fetch is the same under S' and S. If the first hole is red, then under S', this red block is brought into the cache by the red fetch (but it doesn't represent a better eviction choice for the red fetch under S). Thus, the best eviction opportunity at the time of the red fetch is at least as good as that under S', since under S the red fetch occurs later than under S' and thus at a cursor position at least as great.

Let the first hole occur at index h_1 and the second at h_2; let the new hole created by the red fetch under S' occur at index h_r. There are two possibilities:

- $h_2 < h_r$. Under S', the red fetch fills h_1 and the blue fetch fills h_2; under S, the blue fetch fills h_1 and the red fetch fills h_2. The red hole created under S is at a position in the request sequence at least as great as h_r, since the cursor position of the red fetch is at least as great as under S'. Under neither sequence does the blue eviction create a new hole in phase i. Thus, the sets of holes remaining in phase i after completing S' and S are the same, or after S one red hole has a greater index than after S'.

- $h_1 < h_r < h_2$. Under S', the red fetch fills h_1 and creates a hole at h_r. This new hole is the first hole at the time of the blue fetch, and thus the blue fetch fills it (leaving h_2 unfilled). Under S, however, the red fetch may be unable to proceed. The blue fetch fills the hole at h_1; after this, the first hole is at h_2. The red eviction of h_r would violate the rule *do no harm*. But the end result is the same as it is under S' (ignoring holes beyond the end of the phase): the next hole is at h_2, and a new blue hole has been created beyond the end of the phase. The red block requested at h_r does not need to be evicted and then fetched back. (Again, under S it may be possible to create a red hole with greater index; in this case, h_2 gets filled, and the holes phase dominate those after S' by clause 2 of the phase domination lemma.)

□

Lemma 14 *Suppose that fetch sequence S within phase i is obtained from fetch sequence S' by an overlapping swap. Then the set of holes reached by performing S strongly dominates (and thus phase dominates) the set of holes reached under S'.*

Proof: Neither fetch affects the eviction opportunities of the other, since they overlap and evict to different disks. For each of the two fetches under S', the fetch of the same color under S is initiated at a cursor position at least as great. An argument similar to the proof of lemma 13 finishes the proof. □

Lemma 15 Reverse aggressive*'s sequence of fetch opportunities can be obtained from the sequence leading to $A_{n+b-d+1}$ (i.e.* opt*'s sequence of fetches) via a sequence of busy-early swaps, overlapping swaps that don't involve fetches performed by the busy disk, and insertions of extra fetches not matched to any fetch of* opt.

Proof: First we show that for each disk other than the busy disk, any inversion of fetches on that disk and the busy disk is in the "right direction" (i.e. corresponds to a busy-early swap). Let blue denote the color of the busy disk, and let red denote the color of another disk. We refer to fetches using the blue disk as blue fetches (even though blue is the color of the evicted block; the block fetched may be any color), and those using the red disk as red fetches (even though red is the color of the evicted item). For $j \leq b$, let t_{B_j} be the time at which *reverse aggressive*'s j^{th} blue fetch is initiated, and for $j \leq b-d+1$, let t'_{B_j} be the time at which *opt*'s j^{th} blue fetch is initiated. Similarly define t_{R_j}, for $1 \leq j \leq r-1$, and t'_{R_j} for $1 \leq j \leq r$ for the red fetches, where r is the number of red fetches completed by *opt* at or before $T'_i + T_O(\ell)$. First, consider all of *reverse aggressive*'s blue and red fetches except its last $d-1$ blue fetches, and all of *opt*'s blue and red fetches except its last red fetch (which is matched to one of *reverse aggressive*'s last $d-1$ blue fetches). We have that for all $j \leq b-d+1$, $t_{B_j} \leq t'_{B_j}$ (i.e. *reverse aggressive*'s j^{th} blue fetch is no later than *opt*'s) and for all $j \leq r-1$, $t_{R_j} \geq t'_{R_j}$ (i.e. *reverse aggressive*'s j^{th} red fetch is no earlier than *opt*'s). Suppose that there is an inversion in the "wrong direction," i.e. that for some j and some k, $t'_{B_j} < t'_{R_k}$ and $t_{R_k} < t_{B_j}$. Then

$$t'_{B_j} < t'_{R_k} \leq t_{R_k} < t_{B_j} \leq t'_{B_j}$$

which is a contradiction, since $t'_{B_j} < t'_{B_j}$.

Finally, consider one of *reverse aggressive*'s last $d-1$ blue fetches and *opt*'s last (r^{th}) red fetch that it matches. For this blue fetch to be involved in an inversion in the wrong direction means that for some k, $t'_{R_r} < t'_{R_k}$ and $t_{R_k} < t_{B_b}$. Since *opt*'s r^{th} red fetch is its last, the first inequality is false for all $k \leq r$.

For fetches other than blue fetches (i.e. non-busy-disk fetches), let t'_1 and t'_2 be the times of two fetches of *opt* where $t'_1 \leq t'_2$, and let t_1 and t_2 be the times of *reverse aggressive*'s matching fetch opportunities. If *opt*'s fetches don't overlap, then $t'_1 \leq t'_2 - F$. By the definition of matching fetch opportunites, we have $t_1 \leq (T_i - T'_i) + t'_1 + dF - 1$ and $t_2 \geq (T_i - T'_i) + t'_2 + (d-1)F$. Putting these together, we have $t_1 < t_2$, i.e. *reverse*

aggressive's matching fetch opportunities occur in the same order as *opt*'s fetches.

That the cursor positions of the swapped pairs satisfy the inequalities in the definitions of busy-early-swaps and overlapping swaps, respectively, can be seen from the definition of matching fetch opportunities. □

We now complete the proof of lemma 12 using the previous lemmas; recall that we have already shown that $A_{n+b-d+1}$ phase-dominate *opt*'s set of holes $O_{n+b-d+1}$. Lemmas 13, 14, and 15, along with the transitivity of phase domination, together imply that *aggressive*'s holes at time $T_i + t_A(\ell) - 1$ phase dominate *opt*'s holes at time $T'_i + t_O(\ell)$. Thus we have

Corollary 16 Reverse aggressive*'s first hole at time* $T_i + t_A(\ell) - 1$ *is at a cursor position at least as great as* opt*'s first hole at time* $T'_i + t_O(\ell)$.

This contradicts the hypothesis that *reverse aggressive* stalls at time $T_i + t_A(l) - 1$ and *opt* does not stall at time $T'_i + t_O(l)$. □

We now use lemma 12 in order to prove the outer inductive step.

Let f'_j be the j^{th} fetch *opt* performs that completes after time T'_i and at or before time T'_{i+1}, and suppose it begins at time $T'_i + t'_j$. We know that $H^+_{agg.}(T_i)$ strongly dominates $H^-_{opt}(T'_i)$. Therefore, we have that at the time that *opt* initiates f'_1, *reverse aggressive*'s holes strongly dominate *opt*'s.

Define the j^{th} *matching fetch* to be the fetch (if any) that *reverse aggressive* performs on the same disk as f'_j that is initiated in the time interval

$$[T_i + t'_j + (d-1)F, T_i + t'_j + dF - 1],$$

say at time $T_i + t_j$. (Notice this is a different matching than that used in lemma 12. In this matching, fetches of all colors are matched in the same way non-busy-disk fetches were matched in lemma 12.) By lemma 12, we know that *reverse aggressive*'s cursor position at time $T_i + t_j$ is at least as great as *opt*'s cursor position at time $T'_i + t'_j$.

By the same argument as in the last part of the proof of lemma 15, *reverse aggressive*'s sequence of fetch opportunities can be obtained from *opt*'s sequence of fetches by a sequence of overlapping swaps and insertions. Applying the domination lemma, lemma 14, and transitivity of strong domination as needed, we obtain that *reverse aggressive*'s holes after completing its matching fetch opporunities dominate *opt*'s holes after completing its sequence of fetches.

Let c be the color of *reverse aggressive*'s busy disk in phase $i+1$. Consider the fetch f'_r that *opt* has in progress, if any, on its disk c at the time its cursor position first reaches phase $i+1$. Define T'_{i+1} to be time at which this fetch completes, $T'_i + t'_r + F$. T_{i+1} is defined as the time at which *reverse aggressive*'s matching fetch f_r completes; note $T_{i+1} \leq T_i + t'_r + (d+1)F - 1$. If *reverse aggressive* has no matching fetch f_r, then we take T_i to be $T_i + t'_r + dF$. If *opt* has no fetch in progress when its cursor reaches phase $i+1$, let T'_{i+1} be the time at which *opt*'s cursor reaches phase $i+1$. *Reverse aggressive* reaches phase $i+1$ after losing at most $(d-1)F$ steps to *opt* since the start of phase i, and any fetch in progress on disk c completes within another $F-1$ steps; define the completion time of that fetch (if any) as T_{i+1}; if there is no such fetch, let T_{i+1} be equal to $T_i + (d-1)F$. By the preceding argument, the invariants of the outer induction are true for phase $i+1$. □

Conservative: Lower Bound

The following example shows that for $d < F$, there are arbitrarily long strings on which *conservative* requires time $1 + d\frac{K-F}{K}\frac{F}{F+d}$ times the optimal elapsed time.

Example: Suppose that F divides K, and also that d divides K, and consider a repeated cycle on $K + (\frac{K}{F} - 1)d$ blocks. *Conservative* always evicts the page just referenced whenever it fills a hole, since that is the page that won't be needed again for the longest time. Thus *conservative* will never be able to overlap prefetches with each other or with references. Since there are at least $(\frac{K}{F} - 1)d$ holes on each pass through the cycle, *conservative* will spend at least $K + (\frac{K}{F} - 1)d + (\frac{K}{F} - 1)dF$ steps on each pass through the cycle. Suppose that the blocks are colored such that each contiguous sequence of d blocks in the cycle contains one block from each of the d disks. It is not hard to see that *opt* is able to maintain its holes in groups of d, one of each color, spaced F steps apart. Thus *opt* can service the entire sequence without stalling, and requires only $K + (\frac{K}{F} - 1)d$ steps on each pass through the cycle. The ratio of these two expressions (after a little manipulation) turns out to be at least as great as the stated bound.

Conservative: Upper Bound

Theorem 17 *On any reference string R, the elapsed time of* conservative *with d disks on R is at most $d+1$ times the elapsed time of the optimal prefetching strategy on R.*

Proof: Let m be the minimum number of fetches (which is exactly how many fetches *conservative* performs) on request sequence R. *Conservative*'s running time is at most $|R| + mF$, even if it never overlaps prefetches with each other or with the servicing of requests. Since the optimal algorithm *opt* must perform at least as many fetches as *conservative*, and also must

service the request sequence R, *opt*'s running time is at least $\max(|R|, mF/d)$. The ratio of these is maximized with $|R| = mF/d$, and has the value $d+1$. □

Aggressive: Lower Bound

The following example shows that for two disks, there are arbitrarily long strings on which *aggressive* requires time $2 - \frac{4}{F+2}$ times the optimal elapsed time (within an additive constant that depends only on F and K). In general, our bound is a little weaker: for d disks, there are arbitrarily long strings on which *aggressive* requires time $d - \frac{3d(d-1)}{F+3(d-1)}$ times the optimal elapsed time (within an additive constant that depends only on F and K). Consider the sequence

$$b_1 b_2 r_1 \cdots r_F b_3 b_4 r_F \cdots r_1 b_2 b_1 r_1 \cdots r_F b_4 b_3 \ldots$$

where all r_i are red and all b_i are blue. Let $K = F + 2$. The initial cache contents are b_1, b_2, and $r_1 \cdots r_F$; there are holes at the first references to b_3 and b_4. Both algorithms service the initial request of b_1 during the first unit of time. *Aggressive* then evicts the block in its cache not referenced for the longest time, b_1, in order to fetch b_3; the optimal algorithm *opt* does the same. At the completion of this fetch, the next hole for both algorithms is at b_4, and the cursor is at the first request of r_F. *Aggressive* immediately evicts the block among those in the cache not used for the longest time, which is now b_2; *opt* evicts r_1 instead. Both algorithms stall for $F - 2$ steps on the hole at b_4. However, *opt* is able to initiate a fetch of its next hole, r_1, evicting b_3, since the hole is red and the fetch in progress is fetching a blue block; *aggressive* is unable to perform a second fetch in parallel because its next hole (b_2) is also blue. Notice that *aggressive* still has no red holes, and thus can complete only one fetch every F steps. ¿From this point on, *opt* is able to create one red and one blue hole in each subsequence of $F + 2$ requests, and can always fill them without stalling, whereas *aggressive* will always create a pair of blue holes, and will require time $2F$ to serve each subsequence of $F + 2$ requests, since it takes this long to complete two fetches. Thus from this point on, the ratio of *aggressive*'s running time to that of *opt* is $\frac{2F}{F+2} = 2 - \frac{4}{F+2}$.

We have illustrated the case $K = F + 2$, $d = 2$ for simplicity. It is easily generalized to arbitrarily large values of $\frac{K}{F}$ (which are the cases of interest in practice) as follows: let $K = iF + 2$, and interleave i distinct subsequences of F distinct red blocks each with $i + 1$ distinct pairs of blue blocks in round-robin fashion, reversing each subsequence of red blocks and each pair of blue blocks on alternate occurrences. It is not hard to see that *aggressive* will behave similarly to the illustrated case, and that *opt* is able to service the sequence without stalling (after an initial startup period).

The generalization to $d > 2$ is also straightforward. Consider the sequence

$$b_1 \cdots b_d b_1 \cdots b_{d-2} x_1 \cdots x_{d-1} r_1 \cdots r_{F-d+1} x'_1 \cdots x'_{d-1} \cdots$$
$$\cdots b_{d+1} \cdots b_{2d} b_{d+1} \cdots b_{2d-2} \cdots$$

where $F > d$ and $K = F + 2d - 1$, the colors of the b_i are all the same, the colors of the x_i are distinct from each other and the color of the b_i, and the color of x'_i is the same as that of x_i. We omit the details of the startup period, and note that if *aggressive* has holes at $b_1 \cdots b_d$, it will fill them by evicting $b_{d+1} \cdots b_{2d}$ and thus requires time at least dF to serve the sequence up to b_{d+1}. Its state is then similar to the state in which it started, and thus the process can repeat indefinitely. *opt*, on the other hand, is able to maintain d holes of d distinct colors, and can serve the sequence without stalling. Each sequence of $3(d-1)+F$ requests requires time $3(d-1)+F$ for *opt*, and dF for *aggressive*, for a ratio of

$$\frac{dF}{3(d-1)+F} = d - \frac{3d(d-1)}{F+3(d-1)}.$$

Again, generalizing to arbitrary K/F is easy.

Aggressive: Upper Bound

First we state a very simple lemma, leaving the proof to the reader.

Lemma 18 *If a set A of holes dominates a set B of holes, and some hole in A is filled and some hole at a larger index added to A, the resulting holes A' dominate B.*

Theorem 19 *On any reference string R, the elapsed time of* aggressive *with d disks on R is at most $d(1 + \frac{F+1}{K})$ times the elapsed time of the optimal prefetching strategy on R.*

Proof:

In the analysis of aggressive prefetching with one disk, it was shown that if A's holes dominate B's holes, and A's cursor position is at least as great as B's, and each algorithm initiates a fetch, A's holes will continue to dominate B's when the fetch is completed. This result was referred to as the *domination lemma* [8]. The proof of this is similar to but simpler than that of lemma 8 for algorithms working with the reverse sequence.

In order to apply this lemma to more than one disk, we must be sure that when we are comparing a fetch A initiates to a fetch B initiates that the hole being filled by A is *the first missing hole*. If not, the domination lemma does not hold.

In general, we can not ensure that d parallel prefetches *aggressive* initiates will fill the first d holes,

since some of these holes may be of the same color. However, we do know that by the time *aggressive* completes d prefetches on the same disk, the first d holes that were present (and perhaps others) have been filled.

Therefore, our proof strategy is to run *opt* at $1/d$ times the speed of *aggressive*, so that during each subsequence of time in which *aggressive* fills *at least* its first d holes, *opt* can fill *at most* its first d holes. We will show inductively that at the end of each of these subsequences, *aggressive*'s holes dominate *opt*'s holes. This will imply that *aggressive* can can take only d times as long as *opt* to complete a phase.

Notice that as long as there are holes in the phase containing the cursor, there are blocks in the cache which are not requested before the end of the phase (since the cache holds K blocks and there are only K distinct requests in a phase). Since *aggressive* always evicts a block that is not requested until after the block that replaces it, once its cursor enters a phase, *aggressive* will not create any new holes within the phase. Also, once *aggressive* enters a phase, each disk will initiate a fetch every F steps as long as there are holes of that disk's color remaining in the phase.

We prove the following claim:

For each i up to the number of phases in R, there are times T_i and T'_i such that

- $T_i \leq dT'_i + id(F+1)$;
- $H^-_{agg.}(T_i)$ dominates $H^+_{opt}(T'_i)$;
- *aggressive*'s cursor is in the i^{th} phase of the request sequence at time T_i;
- *aggressive*'s cursor position at time T_i is at least as great as *opt*'s cursor position at time T'_i;
- each of *aggressive*'s disks is either ready to initiate a prefetch or is already filling a hole in phase i, for which *opt* has not yet started filling its matching hole.

The theorem follows from the first part of the claim, since each phase has length at least K, so that *opt*'s running time on each phase is at least K (except for a possibly incomplete final phase, which is served by *aggressive* in at most d times as much time as *opt*).

This claim is proven by induction on i. The basis ($i = 1$) is trivial, since both algorithms start at the beginning of the first phase in the same state, with both disks idle.

For the induction, assume that the claim is true for i.

We first show that for each index j in phase i, *aggressive*'s cursor passes j after at most d times as many steps as *opt*'s cursor takes to pass j. Let $T_i + t_A(j)$ be the time *aggressive* serves request j, and let $T'_i + t_O(j)$ be the time *opt* serves j. Assume by way of contradiction that *aggressive*'s cursor falls behind *opt*'s (relative to the start of the phase) by more than a factor of d, and let j be the least index for which this happens, i.e., $t_A(j) > dt_O(j)$. It must be true that *aggressive* has a hole at j (or equivalently stalls on the j^{th} request in the phase) at time $T_i + t_A(j) - 1$, and that the j^{th} request in the phase is in *opt*'s cache before time $T'_i + t_O(j)$, since $T_i + t_A(j)$ is the *first* time *aggressive*'s cursor falls behind *opt*'s by more than a factor of d. As noted previously, each disk of *aggressive*'s fills a hole every F steps as long as there are holes of that disk's color in the phase. Let h be the number of *aggressive*'s holes at time T_i that are the same color as the one at j, up to and including the one at j. Then $t_A(j) \leq hF$, since the hole at j is filled at a time no later than $T_i + hF$. At time T'_i, *Opt* has at least h holes at or before j, since *aggressive*'s holes at T_i dominate *opt*'s holes at T'_i. Thus the earliest time *opt* could finish filling all its holes up to index j is $T'_i + \lceil h/d \rceil F$, even if it fills a hole every F steps on each disk. Thus we have a contradiction: $hF \geq t_A(j) > dt_O(j) \geq d(\lceil h/d \rceil F) \geq hF$.

To show that *aggressive*'s holes after finishing phase i dominate *opt*'s holes, we need another induction. Let $I'_j = [T'_i + jF, T'_i + (j+1)F)$, $j \geq 0$, and let c_j be *opt*'s cursor position at time $T'_i + jF$. Also, let $I_j = [T_i + t_A(c_j), T_i + t_A(c_j) + dF)$. Consider the set of at most d fetches that *opt* completes during I'_j. We match these to the set of fetches *aggressive* initiates during I_j. We prove by induction on j that *opt*'s set of holes, after completing all its fetches that *complete* in I'_j, is dominated by *aggressive*'s set of holes, after completing all its fetches that are *initiated* in I_j. The base case follows from the fact that $H^-_{agg.}(T_i)$ dominates $H^+_{opt}(T'_i)$ from the inductive hypothesis of the outer induction. For the inductive step, note that each fetch *opt* completes during I'_j is initiated at a cursor position at most c_j, and that *aggressive*'s cursor position is at least c_j during the time period I_j. Thus *aggressive*'s fetches can be matched to *opt*'s and the domination lemma implies that *aggressive*'s resulting holes dominate *opt*'s resulting holes. Any extra fetches of *aggressive* (there may actually be as many as d^2 by *aggressive* and as few as 0 by *opt* during their respective time intervals) don't affect this, by lemma 18. As a special case, if *aggressive* should stop fetching altogether at some time (and thus have fewer than d fetches to match to *opt*'s), we know that *aggressive* has reached the optimal cache configuration: its cache contains the next K distinct requests, and its holes are as far from the cursor as possible. These holes certainly dominate *opt*'s holes at any earlier cursor position.

Consider the value j such that *opt*'s cursor reaches phase $i+1$ during I'_j, and let C be this cursor position.

Then by our inductive hypothesis, *aggressive*'s holes after completing all fetches in the corresponding interval dominate *opt*'s holes after completing all fetches in this interval. Note that the last matching fetch in this set completes by time $T_i + t_A(c_j) + (d+1)F$. Let T_{i+1} be the maximum of the times at which fetches initiated in I_j complete and the time that *aggressive*'s cursor reaches phase $i+1$. Then $T_{i+1} \leq \max(T_i + t_A(c_j) + (d+1)F,\ T_i + t_A(C)) \leq \max(T_i + d(t_O(c_j) + F) + F,\ T_i + dt_O(C))$. If we let $T'_{i+1} = t_O(C)$ be the time at which *opt*'s cursor reaches phase $i+1$, then since $T_i \leq dT'_i + (F+1)di$, we have that $T_{i+1} \leq dT'_i + (F+1)di + d(t_O(c_j) + F) + F \leq dT'_{i+1} + (F+1)d(i+1)$, as needed. □