A Study of Integrated Prefetching and Caching Strategies

Pei Cao * Edward W. Felten* Anna R. Karlin † Kai Li *

Abstract

Prefetching and caching are effective techniques for improving the performance of file systems, but they have not been studied in an integrated fashion. This paper proposes four properties that optimal integrated strategies for prefetching and caching must satisfy, and then presents and studies two such integrated strategies, called *aggressive* and *conservative*. We prove that the performance of the *conservative* approach is within a factor of two of optimal and that the performance of the *aggressive* strategy is a factor significantly less than twice that of the optimal case. We have evaluated these two approaches by trace-driven simulation with a collection of file access traces. Our results show that the two integrated prefetching and caching strategies are indeed close to optimal and that these strategies can reduce the running time of applications by up to 50%.

1 Introduction

Prefetching and caching are two known approaches for improving the performance of file systems. Although they have been studied extensively, most studies on prefetching have been conducted in the absence of caching or for a fixed caching strategy. The interaction between prefetching and caching is not well understood.

The main complication is that prefetching file blocks into a cache can be harmful even if the blocks will be accessed in the near future. This is because a cache block needs to be reserved for the block being prefetched at the time the prefetch is initiated. The reservation of a cache block requires performing a cache block replacement earlier than it would otherwise have been done. Making the decision earlier may hurt peformance because new and possibly better replacement opportunities open up as the program proceeds.

1.1 An Example

Consider a program that references blocks according to the pattern "ABCA". Assume that the file cache holds two blocks, that fetching a block takes four time units, and that A and B are initially in the cache.

As shown in Figure 1, a no-prefetch policy (using the optimal offline algorithm) would hit on the first two references, then miss on the reference to C, discarding B, and finally hit on A. The execution time of the no-prefetch policy would therefore be *eight* time units (one for each of the four references, plus four units for the miss.)

By contrast, as Figure 2 shows, a policy that prefetches whenever possible (while making optimal replacement choices) takes *ten* time units to execute this sequence. After the first successful access to A, a prefetch of C is initiated, discarding A. This prefetch hides one unit of the fetch latency, so the access to C stalls for only three cycles. Once C arrives in memory, the algorithm initiates another prefetch, bringing in A and discarding B, while accessing C. The next reference, to A, stalls for three time units, waiting for the prefetch of A to complete. This algorithm uses ten time units, one for each of the four references, plus two stalls of three time units each.

This example illustrates that aggressive prefetching is not always beneficial. The no-prefetch policy fetched one block, while the aggressive prefetching algorithm fetched two. The price of performing an extra fetch outweighs the latency-hiding benefit of prefetching in this case. On the other hand, prefetching might have been beneficial under slightly different circumstances. If the reference stream had been "ABCB" instead of "ABCA", then aggressive prefetching would have outperformed the no-prefetch policy. Thus we see that aggressive prefetching is a double-edged sword: it hides fetch latency, but it may increase the number of fetches.

This paper takes a first step towards an understanding of the answer to a basic question: given detailed information about file accesses, what is the optimal combined prefetching and caching strategy? We begin by describing four fundamental properties that any optimal strategy must satisfy. We then present two simple strategies with these properties: *aggressive* and *conservative*. We show that for any sequence of file-block accesses, the elapsed time of the *conservative* strategy is within a factor of two of the elapsed time of the optimal prefetching schedule, and that this bound is tight. For *aggressive*, we are able to show a stronger performance bound, which depends on parameters of the specific system, such as the relative cost of fetching a block, and size of the cache. For typical values of these parameters, we show that the elapsed time of *aggressive* is within a factor significantly less than twice that of the optimal prefetching schedule. This bound is also tight.

We have evaluated these two strategies by trace-driven simulations with a collection of file access traces. We have compared these strategies with existing approaches. Our simulation results show that the performance of these two approaches is better than any existing approach, and is indeed close to optimal. Our two policies reduce running time by up to 50% compared to conventional file systems.

2 Basics of Prefetching and Caching

In this section, we explain our model on an intuitive level, and describe four rules that an optimal policy must follow.

*Department of Computer Science, Princeton University, Princeton, NJ. {pc,felten,li}@cs.princeton.edu

†Department of Computer Science, University of Washington Seattle, WA. karlin@cs.washington.edu

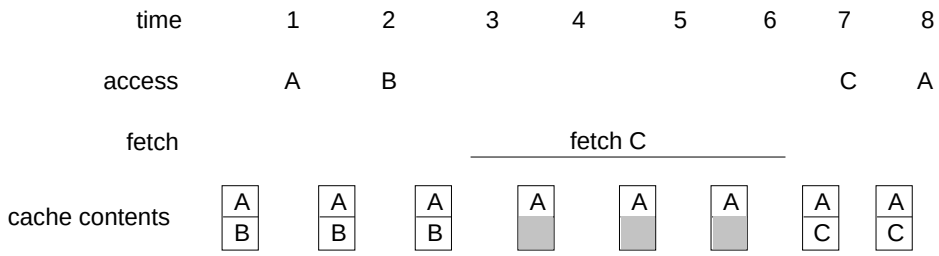

Figure 1: Example under no-prefetch policy. Eight time units are required. The first two references, to A and B hit; the next reference, to C, stalls for four time units while C is fetched and B is discarded. The final reference, to A, hits.

2.1 Model Definition

We consider the execution of a program that makes a known sequence $(r_1, r_2, \ldots, r_n)$ of references to data blocks. The program executes one reference per time unit. The cache can hold k blocks, where $k < n$. If a reference hits in the cache, it can be satisfied immediately; otherwise, the missed block has to be fetched from the backing store.

The system can either fetch a block in response to a cache miss (*on-demand fetch*), or it can fetch a block before it is referenced in anticipation of a miss (*prefetch*). It takes F time units to fetch a block from backing store into cache. At most one fetch can be in progress at any given time. When a fetch is initiated, some block must be discarded from the cache to make room for the incoming block; while the fetch is in progress, neither the incoming block nor the discarded block is available for access.

When the program tries to access a block that is not available in the cache, it stalls until the block arrives in the cache. The stall time is either F if the block is fetched on-demand, or $F - i$ if the fetch was started i time units ago. The total elapsed time for a reference string is the total reference time (or the number of references) plus the sum of stall times.

The goal of a prefetching and caching policy is to make the decisions

- when to fetch a block from disk;
- which block to fetch;
- which block to replace when the fetch is initiated;

so that the total elapsed time is minimized.

Clearly, this models a system with one disk or file server, and it considers only read references. The time unit models the fact that there is generally CPU time spent between two consecutive file references — the CPU time includes the time to copy the accessed file data from kernel address space to user address space buffer, and the time for the application to consume the file data. The model simplifies the real situation by assuming that the CPU time between every two file references (called "reference time") is the same, and is called one time-unit. As we show later in our simulations, this simplification still approximates real systems reasonably well.

We should emphasize: our goal is to find a simple near-optimal *off-line* policy for prefetching and caching that minimizes the total elapsed time of a known sequence of references.

2.2 Four Rules for Optimal Prefetching and Caching

This subsection presents four rules that an optimal prefetching and caching strategy must follow. These rules are mandatory, in the sense that any algorithm can easily be transformed into another algorithm, with performance at least as good, that follows the rules. Thus, the search for optimal policies can be restricted to policies that follow these rules[1].

Correctness of these rules is easily proved; we omit the proofs to simplify the discussion.

Rule 1: Optimal Prefetching *Every prefetch should bring into the cache the next block in the reference stream that is not in the cache.*

Rule 2: Optimal Replacement *Every prefetch should discard the block whose next reference is furthest in the future.*

The first two rules uniquely determine what to do, once the decision to prefetch has been made. However, they say nothing about when to fetch — the next two rules speak on that question.

Rule 3: Do No Harm *Never discard block A to prefetch block B when A will be referenced before B.*

A prefetch that disobeys this rule does more harm than good — it can only increase the program's running time. Unfortunately, existing prefetching algorithms do not always satisfy this requirement, because they separate caching from prefetching, and separate cache replacement decisions from prefetching decisions.

Rule 4: First Opportunity *Never perform a prefetch-and-replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.*

The algorithm must perform each operation at the first opportunity. A new opportunity may arise when either
(a) a fetch completes, or
(b) the block that would be discarded (according to Rule 2) was just referenced in the previous time unit.

Note that condition (b) is the only circumstance under which Rule 2 can change its recommendation about which block to discard.

[1] These rules hold for the situation we are modeling — filesystem caching on a system with one disk. They may not apply to all situations where prefetching is done; in particular, they do not apply to hardware prefetching, where the cache has limited associativity. We believe, however, that variations of these rules can be formulated for other situations.

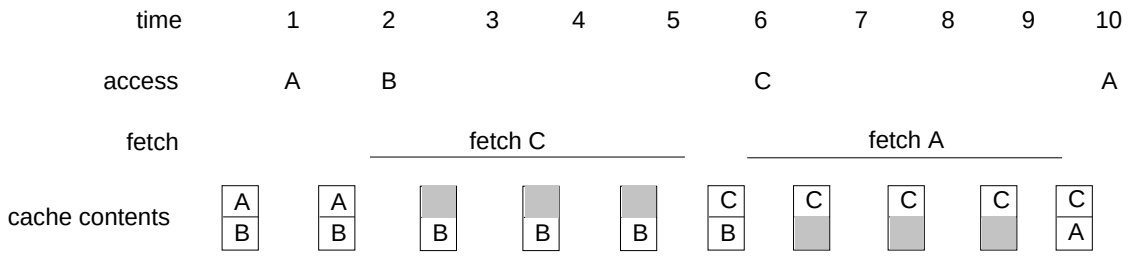

Figure 2: Example under always-prefetch policy. Ten time units are required. The first reference, to A, hits. Concurrently with the next reference (to B), a prefetch of C is initiated, discarding A. The reference to C stalls for three time units, waiting for the prefetch to finish. While C is being referenced, another prefetch, for A, is initiated; B is discarded. The last reference, to A, stalls for three time units waiting for the second prefetch to finish.

2.3 Policy Alternatives

Taken together, the four rules provide some guidance about when to prefetch; once a decision to prefetch has been made, they uniquely specify what should be prefetched and what should be discarded.

Thus, we can imagine a prefetching policy as answering a series of yes/no questions, with each question of the form "Should I prefetch now?" As the program executes, a series of opportunities to prefetch arise, and the policy is asked whether to take each opportunity or to let it pass. We now present two policies, *conservative* and *aggressive*. Both satisfy the four properties described in section 2.2.

2.4 The *Conservative* Strategy

The *conservative* prefetching strategy tries to minimize the elapsed time while performing the minimum number of fetches. The *conservative* prefetching strategy performs exactly the same replacements as the optimal offline demand paging strategy MIN, except that each fetch is performed at the earliest opportunity consistent with the four rules in section 2.2.

2.5 The *Aggressive* Strategy

The *aggressive* prefetching strategy is the strategy that always prefetches the next missing block at the earliest opportunity consistent with the four rules of section 2.2. In order to bring in this next missing block, *aggressive* replaces the block whose next reference is furthest in the future. Notice that *aggressive* is not mindlessly greedy — it at least waits until there is a block to replace whose next reference is after the request to the missing block.

Aggressive is the algorithm that always prefetches the next block not in cache at the earliest possible moment, replacing the block whose next request is furthest in the future. Of course, *aggressive* is not mindlessly greedy — it at least waits until prefetching would be profitable, that is, until there is a block in cache whose next reference is after the first reference to the block that will be fetched.

3 Theoretical Analysis

3.1 Problem Definition

We review our model, slightly more formally. The input to a prefetching algorithm is a *reference string* $(r_1, r_2, \ldots, r_n)$ representing the ordered sequence of file-block accesses to be performed. Recall that the cache holds k blocks, that each access takes unit time, and that fetching a block from backing store takes F time units.

We imagine that there is a cursor, which advances at a fixed rate along the reference string until it hits an access, say r_j, to a file block that is not present in the cache. The cursor then stays at reference r_j until the block arrives in cache. If the processor must wait for block r_j to arrive, we say that *the cursor stalls at reference* r_j. The total amount of time the cursor spends stalled is called the *stall time*.

If the prefetch for r_j is initiated when the cursor is at reference r_i, and $i + F \leq j$, then the cursor does not stall at r_j because the block will have already arrived in memory by the time it is referenced. Otherwise, the cursor stalls for $i + F - j$ time units.

We assume that prefetches are serialized; if a prefetch is initiated at time t, then the next prefetch can be initiated no earlier than time $t + F$.

We also assume that if block q is replaced in order to bring block p into the cache, then block q becomes unavailable for access at the moment the prefetch is initiated, and block p becomes available only when the prefetch terminates (F time units later).

Goal of Prefetching Strategy: To minimize the elapsed time of executing the reference string, where the elapsed time is the time to access the n blocks in the reference string (n time units) plus the stall time.

3.2 Summary of Results

We have already described two prefetching algorithms: *aggressive* and *conservative*. Both of these algorithms satisfy the four properties of an optimal prefetching/caching strategy. However, as implied by their names, they lie on opposite points along the spectrum between aggressive and conservative prefetching.

Before studying these algorithms in more detail, it is worth pointing out that the optimal prefetching schedule for a given reference string, i.e. that which minimizes elapsed

time, is computable in a straightforward manner via dynamic programming. Unfortunately, for large k and n, the obvious dynamic program is computationally infeasible. Hence, we focus our attention on bounding and measuring the performance of simple algorithms, such as *aggressive* and *conservative*.

Our main results are the following:
1. On any reference string R, *conservative* performs exactly the same number of fetches as the optimal off-line demand paging algorithm does on R.
2. On any reference string R, *aggressive* performs at most the number of fetches performed by LRU demand paging on R.
3. On any reference string R, and for $F \leq k$, the elapsed time of *aggressive* is at most $(1 + F/k)$ times the elapsed time of the optimal prefetching schedule on R. If $F > k$, the elapsed time of *aggressive* is at most twice that of the optimal prefetching schedule.
4. The previous bound is nearly tight: There is a reference string R' on which the elapsed time of *aggressive* is $1+(F-2)/(k+1)$ times that of the optimal prefetching schedule on R'.
5. On any reference string R, the elapsed time of *conservative* is at most twice that of the optimal prefetching schedule on R.
6. The previous bound is nearly tight: There is a reference string R' on which the elapsed time of *conservative* is $2(1-1/F)$ times that of the optimal prefetching schedule on R'.

All of these results are straightforward with the exception of the third. In the next section, we present some details about these results.

3.3 Bounds on Number of Fetches

We can make the following observations about the two algorithms:

By definition, *conservative* performs exactly the same number of fetches as the optimal demand paging algorithm MIN does. However, its elapsed time can be much smaller, since it may be possible to overlap these fetches significantly with the request sequence, as was the case for the second fetch in the example above. Indeed, it is only in very special circumstances (such as, for example, cyclic reference patterns) that *conservative* will not be able to overlap fetches with references.

It is possible to show that *aggressive* performs at most the number of fetches that LRU paging does. The proof of this is the same as Belady's theorem mutatis mutandis, so we omit the details. This suggests that on reference patterns with a great deal of locality, *aggressive* may have near-optimal performance, since it will not perform an excessive number of fetches, and it will be able to overlap those fetches with references to a great extent.

3.4 Bounds on Elapsed Time

3.4.1 *Aggressive*: Lower Bound

The following example shows that *aggressive* can have an elapsed time which is nearly $1 + F/k$ times that of the optimal prefetching strategy.

Example : Suppose that $F = k - 2$. Suppose that the algorithms start with blocks $b_1, \ldots b_k$ in the cache. The reference string begins with $b_1, b_2, \ldots b_k, b', b_1$. *Aggressive* will begin prefetching b' immediately after the request to b_1, replacing b_1. This prefetch will terminate immediately before the request to b', at which point *aggressive* will begin prefetching b_1.

The optimal decision will have been to prefetch b' after the request to b_2, since in the remainder of the sequence b_2 will never be requested again. Consequently, on the initial part of the sequence *aggressive* incurs $F - 2$ units of stall time on the subsequent prefetch of b_1, whereas the optimal algorithm incurs no stall time by waiting to prefetch b' until b_2 can be replaced. (*aggressive* of course can replace b_2 on the subsequent prefetch of b_1.) Consequently, the optimal algorithm and *aggressive* will both have the same cache state again immediately before each of them references b_1 for the second time.

We can repeat such a subsequence again and again, where each time, by waiting 1 extra time unit, the optimal algorithm needs do one fewer prefetch per $k + 1$ accesses than *aggressive* has to do.

Since the optimal algorithm incurs $k + 1$ time units for every $k + F - 1$ time units incurred by *aggressive*, we get the stated claim. Note that the specific choice of $F = k - 2$ here is not important. For any value of F, it is quite easy to construct reference strings on which *aggressive* has an elapsed time close to $\min(2, 1 + F/k)$ times that of the optimal algorithm. ■

We now show that this is essentially the worst case for *aggressive*.

3.4.2 *Aggressive*: Upper Bound

In what follows, we will need to divide the reference string into *phases*. The first phase starts with the first reference, and ends immediately before the reference to the $(k + 1)$st distinct block. In general, the ith phase ends immediately before the reference to the $(k + 1)$st distinct block in the phase. Note that phases are a property of the reference string and hence the choice of prefetching strategy is irrelevant to their definition.

Theorem 1 *On any reference string R, the elapsed time of* aggressive *on R is at most the elapsed time of the optimal prefetching strategy on R plus F times the number of phases in the reference string.*

A rigorous proof of the theorem is presented in Appendix 1. The idea of the proof, however, is not difficult. Let *opt* be the optimal prefetching strategy. We show inductively that *aggressive* can fall behind *opt* by at most an additional F time units each phase. Consider the first phase. At time 0, both algorithms are missing the same subset of blocks, say i of them, from the first phase. Therefore, both algorithms have in their cache i blocks that will not be referenced in the first phase. Since both algorithms always replace the block whose first reference is farthest in the future, during the first phase neither will ever replace a block that will be referenced in the first phase. Consequently, since *aggressive* prefetches those i blocks at the earliest opportunity, *aggressive* will stay ahead of *opt* at least until the cursor reaches the first reference of the second phase. Here, *aggressive* may stall and *opt*'s cursor may pass it by, since *aggressive* may have prefetched some blocks in the first phase earlier than *opt* and therefore generated missing blocks in the second phase that are earlier than those generated by *opt*. Imagine now that we "stop" *opt* at the phase boundary for F time units. This gives *aggressive* enough time to complete the extra fetch. More importantly, it is possible to show that now once again *aggressive* has its missing blocks in positions that are at least as good as those of *opt* (where "at least as good" means that

aggressive's ith missing block occurs later in the sequence than *opt*'s ith missing block). This is all we need to complete the induction.

The following corollary is then straightforward.

Corollary 2 *On any reference string R, the elapsed time of* aggressive *on R is at most* $\min(1+F/k, 2)$ *times the elapsed time of the optimal prefetching strategy.*

Proof: Suppose that $F \leq k$. Since each phase lasts for at least k references, the number of phases is at most the time spent by *opt* referencing blocks divided by k. For the other case, $F > k$, observe that it follows from the definition of phases that *opt* has to perform at least one fetch per phase. This fetch takes F time units. Since *aggressive* incurs at most one extra fetch per phase, its elapsed time is at most twice *opt*'s. ■

3.4.3 *Conservative*

We have already observed that *conservative* places the minimum possible load on the disk. However, its elapsed time is not generally as good as that of *aggressive*. The following example shows that *conservative*'s elapsed time can be nearly twice optimal.

Example : Suppose that F divides k, that the algorithms start with blocks $b_1, \ldots b_k$ in the cache and that the reference string is a cyclic pattern, $b_1, \ldots b_k, b_{k+1}, b_{k+2}, \ldots, b_{k+k/F}$ repeated many times. It is easy to see that *conservative* will never be able to overlap fetches with references. Since k/F fetches will be done per pass through the cycle, and *conservative* will stall for F time units on each, its elapsed time will be at least $2k$ per pass through the cycle. On the other hand, it is not difficult to see that by keeping the missing blocks at least F references apart, this sequence can be prefetched so that all fetches overlap references, i.e. so that the elapsed time is $k(1 + 1/F)$ per pass through the cycle. ■

It is also straightforward to see that *conservative*'s elapsed time is at most twice optimal.

Lemma 3 *On any reference string R, the elapsed time of* conservative *on R is at most twice the elapsed time of the optimal prefetching strategy.*

Proof: Any prefetching strategy, and in particular *opt*, performs at least as many fetches as *conservative*. Since *opt* performs the same number of references as *conservative* and at best entirely overlaps its fetches with references, *opt*'s elapsed time is at least half of that of *conservative*. ■

4 Simulations

We have measured the performance of *aggressive* and *conservative*, along with several existing algorithms, by trace-driven simulation. We first discuss various existing approaches, then introduce our traces and simulation models, and finally present simulation results.

4.1 Existing Approaches

We have compared the *aggressive* and *conservative* algorithms with six existing algorithms for caching and prefetching. We will call them *LRU-demand*, *OPT-demand*, *LRU-OBL*, *OPT-OBL*, *LRU-sensible*, and *LRU-throttled*.

LRU-demand and *OPT-demand* are demand-paging algorithms. They only fetch a block from disk when it is accessed; in other words, there is no prefetching. *LRU-demand* specifies that the block that is least recently used should be replaced when necessary; *OPT-demand* specifies that the block that will be referenced furthest in the future should be replaced. Traditional file systems use LRU as replacement algorithm, although recent research [4] has shown that with application knowledge it is possible to make replacement decisions that are close to optimal.

LRU-OBL and *OPT-OBL* model the above replacement algorithms with the addition of sequential one-block lookahead (OBL) prefetching. Traditional file systems use OBL to take advantage of the fact that files are often accessed sequentially [1]. OBL prefetches block K+2 of a file whenever the last two references to the file were to block K and block K+1. Existing file systems mostly use the combination of OBL with LRU replacement[2]; we call the resultant algorithm *LRU-OBL*. Combining OBL with optimal replacement yields the *OPT-OBL* algorithm.

The last two algorithms are intended to model approaches taken in recent research projects on prefetching in file systems. Some of these let users or applications provide information about future accesses and use this information to guide prefetching[18], while others try to predict future accesses based on patterns observed in previous accesses[6, 10]. LRU is typically used as the cache replacement algorithm, even when information about the future reference string is available.

One problem with such approaches is "thrashing". Thrashing happens when prefetching decisions are not integrated with caching decisions: more precious blocks are replaced in order to prefetch less precious blocks, or prefetched file blocks are replaced before they are accessed. Both types of mistakes violate the "Do No Harm" rule, and thus hurt performance.

Some approaches use "throttling" — putting an upper limit on the number of blocks that have been prefetched but not yet accessed — to limit the occurrence of "thrashing". Unfortunately, throttling is an *ad hoc* approach that doesn't always work well. The right approach, when the necessary information is available, is to follow the rules discussed in section 2.2.

We simulate two algorithms based on these approaches: *LRU-sensible* and *LRU-throttled*. *LRU-sensible* looks into the future reference string and fetches the next missing block at the earliest possible moment, subject to the "Do No Harm" rule. Among all the blocks that can be replaced for this fetch, it chooses the least-recently-used one. *LRU-sensible* performs the best among all prefetching algorithms that use LRU as replacement principle (the proof is trivial from the "First Opportunity" rule in section 2.2), and we use it to approximate existing prefetching approaches that follow the "Do No Harm" rule.

LRU-throttled models prefetching approaches that do not follow the "Do No Harm" rule but rather use "throttling" or similar measures. Given the sequence of future accesses (e.g. from application hints), the prefetcher of *LRU-throttled* walks down this sequence, allocates a buffer and issues a prefetch for every missing block. It stops when one-third of the cache contains blocks that have been prefetched but haven't been referenced yet[3].

[2] The prefetched block is typically inserted in the most-recently-used end of the LRU list when the fetch finishes.

[3] In [19] 150 blocks is the throttling limit, out of a cache of 400 blocks. Hence in our simulation we set the throttling limit to be

4.2 File Access Traces

We used two sets of traces. We collected one set by tracing several applications running on a DEC 5000/200 workstation under Ultrix 4.3. The other set is from the Sprite file system traces from the University of California at Berkeley [1].

We instrumented our Ultrix 4.3 kernel to collect file access traces from a set of read-dominated applications on which existing file systems perform poorly. We chose the following representative applications:

cscope[1-3]: an interactive C-source examination tool written by Joe Steffen, searching for eight symbols (cscope1) in a 18MB software package, searching for four text strings (cscope2) in the same 18MB software package, and searching for four text strings (cscope3) on a 10MB software package;

dinero: a cache simulator written by Mark Hill, running on the cc trace;

glimpse: a text information retrieval system [13], searching for four keywords in a 40MB snapshot of news articles;

postgres-join: the Postgres relational database system (version 4.0.1) developed at the University of California at Berkeley, performing a join between an indexed 32MB relation and a non-indexed 3.2MB relation (the relations are those used in the Wisconsin Benchmark [9]). Since the result relation is small, most of the file accesses are reads.

The Sprite traces consist of five sets, recording about 40 clients' file activities over a period of 48 hours (traces 1, 2 and 3) or 24 hours (traces 4 and 5). We evaluated the performance of client caching because it is important in a system with a slow network like ethernet. For each client we extracted its file activities to Sprite's main file server.

We eliminated traces that are dominated by cold-start misses, keeping only those for which cold-start misses account for fewer than half of the misses of the LRU algorithm on a 7MB cache[4]. This resulted in 13 client traces[5]. We denote these traces as Sprite[1-13].

The Sprite traces are not read-dominated: about 10% to 30% of the accesses are writes. In our simulations, however, we treated all write accesses as if they are read accesses. Although this is not accurate, we can still get some insights on how the algorithms might perform in practice.

4.3 Simulation Models

Our simulations used two models of file accesses: *simplified* and *realistic*. The *simplified* model corresponds to the theoretical model discussed in previous sections: it assumes that the "reference time" (i.e. the amount of CPU time spent by the program between two consecutive file accesses) is uniform and defined as one time unit, and fetching data from disk takes F time units. The *realistic* model uses the actual time interval between references from the trace, and uses the average (measured) disk access time for the trace as the fetch time. We can simulate only the Ultrix traces under the *realistic* model due to lack of necessary timing data in the Sprite traces. The average reference time and disk access time for each Ultrix trace is shown in Table 1.

one-third of the cache.

Also, *LRU-throttled* attempts to simulate the algorithm described in [19], which is intended for applications that do not reuse their data. *LRU-throttled* also implements a further improvement on this algorithm: moving soon-to-be-used blocks to the most-recently-used end of the LRU list.

[4] 7MB is the average file cache size for the Sprite clients.

[5] client 29, 33, 53, and 81 of trace 1, client 60, 62, 75, 82 of trace 2, client 9, 77, 56, 18 of trace 4, and client 18 of trace 5

		dinero	cscope1	cscope3
Reference Time	Average	11.8	3.13	2.89
	Stddev	0.12	1.07	1.54
Fetch Time	Average	8.7	8.0	11.8
	Stddev	0.38	0.40	0.69
		cscope2	glimpse	pjoin
Reference Time	Average	4.13	1.17	10.4
	Stddev	1.56	3.83	1.27
Fetch Time	Average	14.3	7.6	17.9
	Stddev	0.53	0.95	0.26

Table 1: Average and standard deviation of the reference times and disk fetch times in each Ultrix trace. Times are in milliseconds.

There are still a number of differences between the *realistic* model and the actual file systems: disk access times are not uniform; our simulation ignored meta-data (directories, inodes, etc.) accesses; we assume that disk blocks are allocated in 8KB blocks, whereas real file systems allocate smaller blocks (e.g. 1KB blocks) for small files, hence our simulation tends to inflate file data set size for applications that use lots of small files.

We calibrated our simulator under the *realistic* model against real systems, by comparing the simulated elapsed time and number of disk fetches of *LRU-OBL*, which is the algorithm used in Ultrix 4.3, to the measured elapsed time and number of disk fetches for the Ultrix traces. The results are shown in Table 2. Despite the above differences, the simulated results are usually within 10-15% of the measured ones. Hence, we believe our simulation predicts real performance reasonably well.

We also compared the simulation results under the *simplified* and the *realistic* models for Ultrix traces. We took the average reference time of each trace as one time unit, and set F to be the ratio of the average disk access time to the average reference time. The simulation results under this *simplified* model match those of corresponding simulations under the *realistic* model very well[6] — the differences are within 2%. We believe the *simplified* model predicts real system performance reasonably, and hence that our theoretical studies (which are under the *simplified* model) apply to real systems.

4.4 Simulation Results

We simulated all eight algorithms for both Ultrix traces and Sprite traces. We used the *realistic* model when simulating Ultrix traces, and the *simplified* model when simulating Sprite traces. The simulations were run varying the relative cost of disk fetch time versus the reference time. For the Ultrix traces, we simulated these algorithms with the CPU time reduced by factors of 1, 2, 4, and 8, and reported the elapsed time in seconds.

For the Sprite traces, we simulated setting F to 3, 5, 10 and 20. We use the minimum total fetch time M, i.e. the theoretically minimal number of disk fetches times the fetch time, as a baseline for comparison. For each F, we report the normalized elapsed time (i.e. the ratio between the elapsed time and M).

We report only averages over each set of traces (Ultrix and Sprite) here. The results from individual traces differ

[6] For the simulations under the realistic model, we scaled the disk fetch time to be an integer multiple of the average reference time, corresponding to the constraint that F be an integer.

			dinero	cscope1	cscope2	cscope3	glimpse	pjoin
6.4MB	Elapsed	Simulated	105	70	92	194	101	207
6.4MB	Time	Measured	117	62	96	191	126	225
6.4MB	Disk	Simulated	8807	8518	6353	10769	9480	6624
6.4MB	Fetches	Measured	8888	8634	6576	11785	10435	6706
12MB	Elapsed	Simulated	105	31	92	194	101	183
12MB	Time	Measured	99	28	57	188	113	202
12MB	Disk	Simulated	988	1071	6353	10765	9466	5305
12MB	Fetches	Measured	997	1141	2815	11717	9720	5437

Table 2: Comparision of simulated and measured elapsed time and number of disk fetches for the Ultrix traces with two cache sizes: 6.4MB and 12MB. The differences are within 10-15% except for cscope2 with 12MB cache. (We believe this difference is due to the fact that cscope2 accesses lots of small files, so our assumption that file blocks are allocated in 8KB blocks artificially inflates the file data set size.)

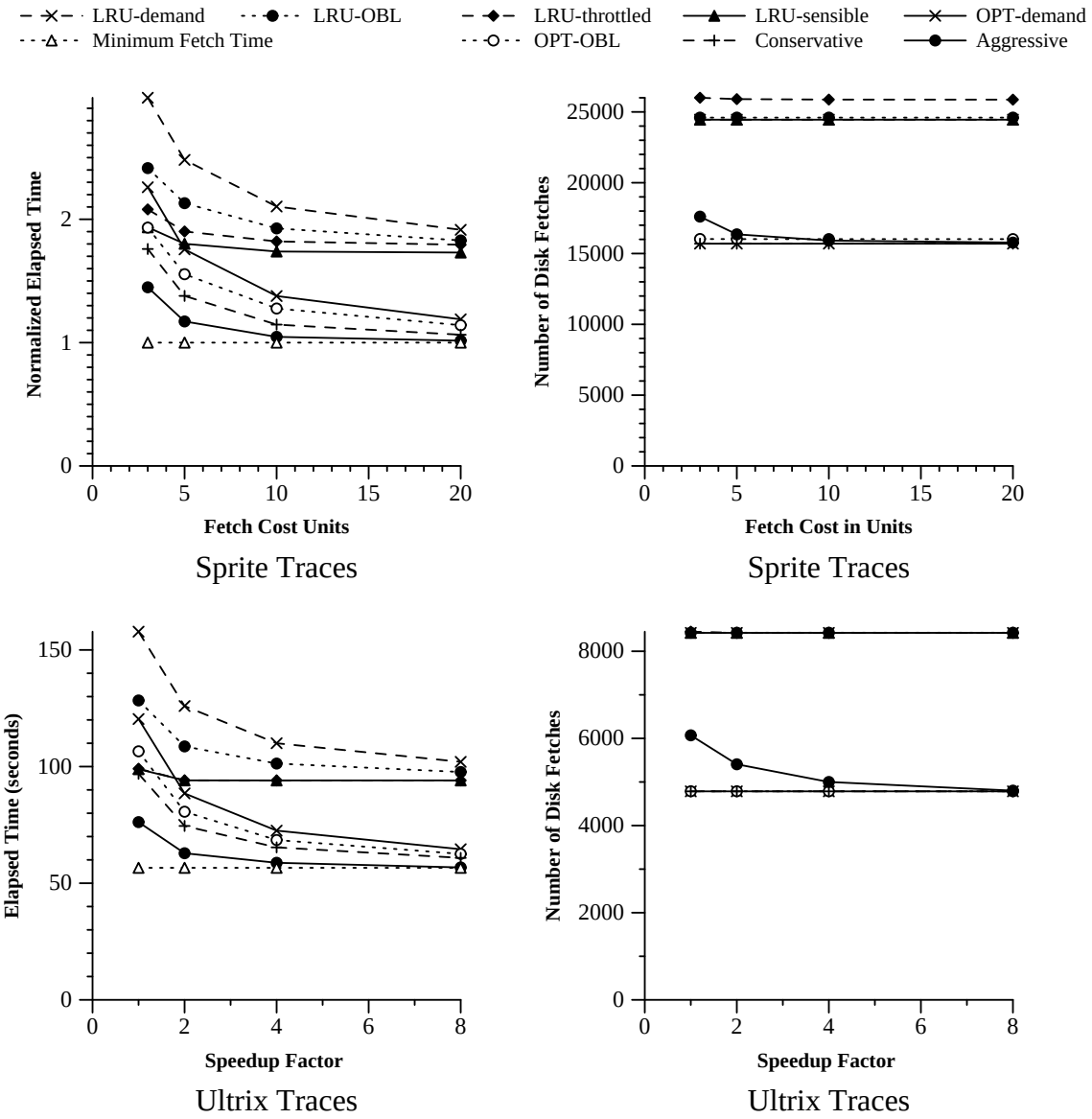

Figure 3: Elapsed time and number of disk fetches: at the top, averaged over the Sprite traces, with cache size 7MB; at the bottom, averaged over the Ultrix traces, with cache size 6.4MB. The elapsed time graphs show the performance of the eight algorithms, as well as the lower bound on the elapsed time given by the minimum fetch time (i.e. the theoretically minimal number of disk fetches times the disk fetch time). In the graphs for numbers of disk fetches, the top group of lines contains *LRU-demand*, *LRU-OBL*, *LRU-throttled*, and *LRU-sensible*, and the bottom group of lines contains *aggressive OPT-OBL*, *OPT-demand*, and *conservative*.

little from the averages.

The averaged predicted running time is shown in Figure 3. These results show that *aggressive* performs the best among all these algorithms, confirming our theoretical results. In general *conservative* does slightly worse. *LRU-sensible* performs the best among all LRU-based algorithms, as expected.

An important observation from these results is that as the relative cost of disk accesses increases, replacement policies make a bigger difference than prefetching policies. This is because the systems become more disk fetch-bound as the relative fetch cost goes up, and the benefit from prefetching (overlapping computation time with fetch time), diminishes as the relative computation time gets smaller.

Therefore, when the fetch cost is large, given any information on future file accesses, the first priority is to make replacement decisions as close to optimal as possible; making good prefetching decisions comes second. The *conservative* algorithm performs well in these cases since it does the minimum number of disk fetches possible.

Of course, this result should not be extrapolated too far. Introducing parallel disks, or increasing the block size, would probably increase the benefit of prefetching. Our results do imply, though, that prefetching alone is of limited value.

Figure 3 also shows the number of disk fetches made by the eight algorithms. As the theory predicts, *LRU-sensible* and *LRU-demand* always do the same number of disk fetches, and *conservative* and *OPT-demand* always do the same number of disk fetches. In addition, *aggressive* does a number of disk fetches between *OPT-demand* and *LRU-demand*.

The number of disk fetches made by *aggressive* decreases as the fetch cost increases. This is due to the self-adjusting property of *aggressive*: as the fetch cost gets larger, *aggressive* cannot prefetch fast enough, so it finds itself prefetching only a few references ahead. Under these conditions *aggressive* makes replacement choices close to those of the optimal replacement algorithm.

The results also show the weakness of the approaches that do not follow the "Do No Harm" rule. For the Sprite traces, *LRU-throttled* makes noticeably more disk fetches than LRU demand paging. This is because *LRU-throttled* ignores the "Do No Harm" rule, and "throttling" does not always eliminate the "thrashing" problem. *LRU-sensible*, on the other hand, always performs better than *LRU-throttled* and does fewer disk fetches. All LRU-based algorithms do about the same number of disk fetches on Ultrix traces because these traces have such poor temporal locality that LRU replacement misses most of the time.

Finally, Theorem 1 says that the running time of *aggressive* is at most $1+F/p$ times that of optimal on any reference stream, where p is the average phase length. Measuring the phase length of the traces, we find that in our traces *aggressive*'s elapsed time is at most 1.024 times that of optimal for the Ultrix traces (as F varies) and at most 1.02 times that of optimal for the Sprite traces.

5 Related Work

Caching has been studied extensively in the past and there is a large body of literature on caching ranging from theory [2, 8], to architecture [21] to file systems [11, 16, 4], etc.

Prefetching has also been studied extensively in various domains, ranging from, uni-processor and multi-processor architectures [20, 5, 3, 22, 24], to file systems [19, 10, 23] to databases[6, 17] and beyond. Sequential one-block lookahead was first proposed in [22]. Few of these studies considered the interaction between prefetching and caching.

In file systems, perhaps the most straightforward approach to prefetching is using large I/O units (i.e. blocks), as in extent-based or similar file systems [14]. However, this approach and one-block-lookahead are often too limited and only benefit applications that make sequential references to large files [10].

Recently there have been a number of research projects on prefetching in file systems. Patterson's Transparent-Informed Prefetching [19] showed that prefetching using hints from applications is an effective way of exploiting I/O concurrency in disk arrays. Griffioen and Appleton's work [10] tries to predict future file accesses based on past accesses using "probability graphs", and prefetch accordingly. These papers demonstrated the benefits of prefetching. However they did not address the interaction between caching and prefetching and did not investigate the combined cache management problem.

There have also been many studies focusing on how to predict future accesses from past accesses. Tait and Duchamp's work [23] tries to detect user's file access patterns and exploit the patterns to prefetch files from servers. Palmer and Zdonik's work on Fido [17] tries to train an associative memory to recognize access patterns in order to prefetch. Vitter and Krishnan's work [6] tries to use compression techniques to predict future file accesses from past access history. All these studies had promising results with respect to prediction. However, the models used assumed that many block I/Os can be done in a time step (i.e. at each reference), which unfortunately is not realistic for file systems. With such a model, cache management becomes substantially less important.

There are a number of papers on prefetching in parallel I/O systems [7, 25]. Although our work focuses on prefetching with a single disk or server, the "Do No Harm" and "First Opportunity" principles apply to prefetching algorithms in the parallel context as well. We believe these principles are important to avoid the thrashing problem [25].

Prefetching in uni-processor and multi-processor computer architectures is similar to prefetching in file systems. However, in these systems there is little flexibility in cache management, as the cache is usually direct-mapped or has very limited associativity. In addition, it is not possible to spend more than a few machine cycles on each prefetch. File systems, on the other hand, can change their cache management algorithms freely and can spare more cycles for calculating a good replacement or prefetching decision, as the potential savings are substantial. On the other hand, Tullsen and Eggers [24] showed that thrashing is a problem when prefetching in bus-based multiprocessor caches, suggesting that the "Do No Harm" rule applies in those systems as well.

Recent work on update policies [15] (i.e. policies on write-backs of dirty blocks) is directly related to our work on prefetching. Although we did not address the write-back problem in this paper, we are working on algorithms for it. Finally, we note that the stashing approach in mobile computing environment [12] is similar to prefetching, although the purpose is quite different: stashing is concerned with the availability of files, while prefetching is more concerned with latency hiding.

6 Conclusions and Future Work

This paper presents a theoretical study and performance evaluation via simulation of two prefetching strategies that

address the interaction between caching and prefetching: the *aggressive* and *conservative* strategies. We have shown that the performance of *aggressive* is always within a factor $\min(1+F/k, 2)$ of optimal, and that the performance of *conservative* is always within a factor of two of optimal. Our simulations with several file access traces from real applications show that these two approaches are indeed close to optimal. In fact, their performance on real traces is significantly better than the worst-case performance ratios given by the theoretical results. Compared with the prefetching and caching methods implemented in most existing file systems, these two strategies can reduce the elapsed times of the applications by up to 50%.

Several questions deserve further study. First, our results are based on full knowledge of file accesses. Although this is unrealistic, we feel that understanding the off-line case is a necessary step towards a full understanding of the on-line case. We can draw an analogy with Belady's famous result [2] showing that MIN is the optimal off-line demand paging strategy. In fact, we have not solved the off-line problem: one direction for future research is to either find a polynomial time algorithm for computing the optimal offline prefetching and caching strategy or to show that the problem is NP-complete.

Having information about the reference patterns may not be entirely unrealistic. Implementations of such strategies can help determined users who are willing to provide the operating system with detailed file access information. Nonetheless, an obvious next step is to investigate the impact of these strategies on applications without perfect knowledge of file accesses.

Our theoretical model does not distinguish write accesses from read accesses. In reality, writes are different from reads in that full-block writes may not need to bring the block into the cache. Treating write accesses differently from reads under a more complicated model may produce more accurate results.

Finally, our models and simulations should be validated by doing a real implementation.

References

[1] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–211, October 1991.

[2] L. A. Belady. A study of replacement algorithms for virtural storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[3] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[4] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. Technical report, Department of Computer Science, Princeton University, June 1994.

[5] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. Also available as U. Washington CS TR 92-06-03.

[6] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[7] Carla Schlatter Ellis and David Kotz. Prefetching in file system for MIMD multiprocessors. In *1989 International Conference on Parallel Processing*, pages 306–314, August 1989.

[8] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., 1973.

[9] Jim Gray. *The Benchmark Handbook*. Morgen-Kaufman, San Mateo, Ca., 1991.

[10] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[11] 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*, pages 6(1):51–81, February 1988.

[12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, pages 6(1):1–25, February 1992.

[13] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Conference Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, January 1994.

[14] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *1991 Winter USENIX*, pages 33–43, 1991.

[15] Jeffrey C. Mogul. A better update policy. In *Proceedings of 1994 Summer USENIX*, pages 99–111, June 1994.

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

[17] 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.

[18] Hugo Patterson, Garth Gibson, and M. Satyanarayanan. Transparent informed prefetching. *ACM Operating Systems Review*, pages 21–34, April 1993.

[19] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proc. Third International Conf. on Parallel and Distributed Information Systems*, September 1994.

[20] Anne Rogers and Kai Li. Software support for speculative loads. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[21] Alan J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[22] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[23] Carl D. Tait and Dan Duchamp. Detection and exploitation of file working sets. Technical Report CUCS-050-90, Computer Science Department, Columbia University, 1990.

[24] Dean M. Tullsen and Susan J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 1993 International Symposium on Computer Architecture*, pages 278–288, May 1993.

[25] Kun-Lung Wu, Philip S. Yu, and James Z. Teng. Performance comparision of thrashing control policies for concurrent mergesorts with parallel prefetching. In *Proceedings of 1993 ACM SIGMETRICS*, pages 171–182, May 1993.

Appendix 1 : The Proof of Theorem 1

Preliminaries:

Let $c_A(t)$ be the index of the next reference at time t when running prefetching algorithm A. Let $H_A(i)$ be the set of blocks *not* present in the cache when the next reference is r_i and we are running prefetching algorithm A. Let $h_A(i,j)$ be the block in $H_A(i)$ whose first occurence after r_i is jth. We will subsequently refer to $h_A(i,j)$ as *A's jth hole*, when the context is clear. Note that j varies between 1 and $n-k$.

Given two prefetching algorithms A and B, we say that *A's cursor at time t dominates B's cursor at time t'* if $c_A(t) \geq c_B(t')$. We also say that *A's holes at time t dominate B's holes at time t'* if $h_B(c_B(t'),j) \leq h_A(c_A(t),j)$ for each j. Finally, we say that *A's state at time t dominates B's state at time t'* if A's cursor at time t dominates B's cursor at time t' and A's holes at time t dominate B's holes at time t'.

The following lemma will be useful.

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

Proof: Suppose that A's holes (the first occurences of each of the blocks not in the cache) at time t are at cursor positions $a_1 < a_2 < \ldots < a_{n-k}$, and that B's holes at time t' are at cursor positions $b_1 < b_2 < \ldots < b_{n-k}$. Since A's state dominates B's, we have that $a_i \geq b_i$ for each i. Suppose that in order to prefetch b_1, B replaces a block b whose first occurence between b_j and b_{j+1}, and that in order to prefetch a_1, A replaces a block a whose first occurence is between a_r and a_{r+1}. If a'_i (resp. b'_i) is the position of A's (resp. B's) ith hole after the prefetch, then for $i \leq r$, $a'_i = a_{i+1}$, $a'_r = a$, and $a'_i = a_i$, for $i \geq r+1$. Similarly, for $i \leq j$, $b'_i = b_{i+1}$, $b'_j = b$, and $b'_i = b_i$, for $i \geq j+1$. In order to show that domination is preserved, we must show that $a'_i \geq b'_i$ for all i at time $t+F$ (resp. $t'+F$) for A (resp. B).

Trivially, for $2 \leq i \leq \min(r,j)$ and $i \geq \max(j+1, r+1)$, $a_i \geq b_i$.

If $r \geq j$, we have $a > a_r \geq b_r$, $a_r > a_{r-1} \geq b_{r-1}$,..., $a_{j+2} > a_{j+1} \geq b_{j+1}$, and $a_{j+1} \geq b_{j+1} > b$, which are the remaining inequalities needed.

If $r < j$, we must show that $a_j \geq b$, $a_{j-1} \geq b_j$, ... $a_{r+1} \geq b_{r+2}$, $a \geq b_{r+1}$. Suppose that one of these inequalities is violated. Consider the largest index that violates the condition. Then we have $a'_{i-1} < b'_{i-1} < b'_i \leq a'_i$ for some $r+1 \leq i \leq j+1$, where a, the replaced block satisfies $a \leq a'_{i-1}$. But this means that the block at cursor position b'_{i-1} was in A's cache at the time it issued the prefetch, and it's first occurence was later than that of a, the block it chose to replace, which is a contradiction to the assumption that A always replaces the block whose first reference is furthest in the future. Therefore, none of these inequalities is violated, and hole domination is preserved.

As for the cursor, if A's cursor stalls, then it does so at A's first hole. But in that case, either B's cursor also stalls at B's first hole or it doesn't reach B's first hole. In either case, B's cursor remains behind A's. ■

Recall that we divided the reference string into *phases* as follows. The first phase starts with the first reference and ends immediately before the reference to the $(k+1)$st distinct block. In general, the ith phase ends immediately before the reference to the $(k+1)$st distinct block in the phase. We are now ready to present the proof of Theorem 1.

Theorem 1 On any reference string R, the *aggressive* prefetching strategy has an elapsed time which is larger than optimal by an additive constant which is at most F times the number of phases in the reference string.

Proof: Let *opt* be the optimal prefetching strategy. We prove the theorem using the following invariant by induction on the number of phases.

Invariant: During phase i, there is a time t such that *aggressive*'s holes dominate *opt*'s holes at time $t' \geq t - iF$ and *aggressive*'s cursor dominates *opt*'s cursor and neither *aggressive* nor *opt* are in the middle of a prefetch at those times.

The proof is by induction on the number of phases. The base case is trivial, since at time 0, both algorithms are in exactly the same state.

Suppose that the invariant is true in phase i, i.e. there is a time t in phase i such that *aggressive*'s holes dominate *opt*'s holes at time $t' \geq t - iF$.

The first observation is that from time t until the first time in phase $i+1$ during which *aggressive* is not initiating a prefetch, *aggressive* never evicts any block that will still be referenced in phase i. Indeed, if *aggressive* is prefetching a block that will still be requested in phase i, then there is some block in the cache which is not requested in phase i. On the other hand, if *aggressive* is prefetching a block p that will not be requested until phase $i+1$, then since it always replaces a block whose first occurence is later than that of the block being prefetched, no new hole can be created within phase i.

This observation implies that at all times $T > 0$, such that *aggressive* is still in phase i at time $t+T$, $c_a(t+T) \geq c_{opt}(t'+T)$, where $c_a(t)$ is the cursor position of *aggressive* at time t. The argument is that the set of holes *aggressive* has within phase i at time t dominates the set of holes *opt* has within phase i at time t', and no new holes are added within the phase. Since *aggressive* prefetches as aggressively as possible, *aggressive* eliminates those holes at the earliest possible time, and therefore, *aggressive*'s cursor can not fall behind *opt*'s cursor within the rest of the phase.

Let $t+t_0, t+t_1 \ldots, t+t_j$ be the cursor positions at which *aggressive* initiates prefetches, after time t but within phase i. If *opt* is in the middle of a prefetch at time $t'+t_l$, then let $t'+t'_l$ be the time at which that prefetch was initiated, otherwise let $t'_l = t_l$. We prove by induction on l that *aggressive*'s state at time $t+t_l$ dominates *opt*'s state at time $t'+t'_l$. The base case is easy. *Aggressive*'s state at time t dominates *opt*'s state at time t'. If $t_0 > 0$, then *aggressive* has nothing that it can prefetch between time t and time $t+t_0$, and hence is in the best possible state during this time. Consequently, its state at time $t+t_0$ dominates *opt*'s state at time $t'+t'_0$. Suppose by the inductive hypothesis that the claim is true for l. Then by the Domination Lemma, *aggressive*'s state at $t+t_l+F$ dominates *opt*'s state at $t'+t'_l+F$. We consider two cases:

1. $t+t_{l+1} = t+t_l+F$: Since $t'_{l+1} \leq t_{l+1}$, $c_a(t+t_{l+1}) \geq c_{opt}(t'+t_{l+1}) \geq c_{opt}(t'+t'_{l+1})$. Furthermore, since $H_{opt(}t'+t'_{l+1}) = H_{opt}(t'+t'_l+F)$, *aggressive*'s state at $t+t_{l+1}$ dominates *opt*'s state at $t'+t'_{l+1}$.

2. $t+t_{l+1} > t+t_l+F$: In this case *aggressive* stops prefetching for a while, which means that its holes are in an optimal state (i.e. there are k distinct requests to the k blocks in the cache prior to any request to a hole). Furthermore, its cursor cannot stall during such a period. Therefore, at best during the period between $t+t_l+F$ and $t+t_{l+1}$, *opt* can get into the same state as *aggressive*.

By the same argument, *aggressive*'s state at time $t+t_j+F$ dominates *opt*'s state at time $t'+t'_j+F$. Since *aggressive* is in phase $i+1$ at time $t+t_j+F$, and $t'+t'_j+F \geq t'+t_j$, we have that at time $T = t+t_j+F$ in phase $i+1$, *aggressive*'s state dominates *opt*'s state at time $t'+t_j \geq t-iF+t_j \geq T-(i+1)F$.

Finally, from the invariant it follows that if *aggressive* finishes processing the sequence at time T, *opt* cannot finish before time T − (number of phases)F, which completes the proof of the theorem. ■