Integrated Parallel Prefetching and Caching

Abstract

Recently there has been a great deal of interest in prefetching from parallel disks, as a technique for enabling serial applications to improve I/O performance. Studies have also shown that for optimal performance, it is important to properly integrate prefetching and caching. In this paper, we study integrated prefetching and caching strategies for multiple disks. We present two algorithms, *regular aggressive* and *reverse aggressive*, and show that *reverse aggressive* is close to optimal. Using trace-driven simulation on a collection of file access traces, we evaluated these algorithms under a variety of data placement alternatives.

Our results show that both algorithms can achieve near linear speedup when the load is distributed evenly on the disks, and *reverse aggressive* performs well even when the placement of blocks on disks distributes the load unevenly. Our simulations also show that, for file system traces, replicating data, even across all of the disks, offers little performance advantage over a striped layout if prefetching is done well. Finally, we evaluate four online variations of the algorithms and show that the online algorithms perform well even with moderate advance knowledge of future file accesses.

1 Introduction

Recent advances in technology have made 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 advantage of both increased bandwidth and reduced contention on individual disk arms. However, many applications do not benefit from the parallelism either because they mostly access small files or because they perform non-sequential accesses.

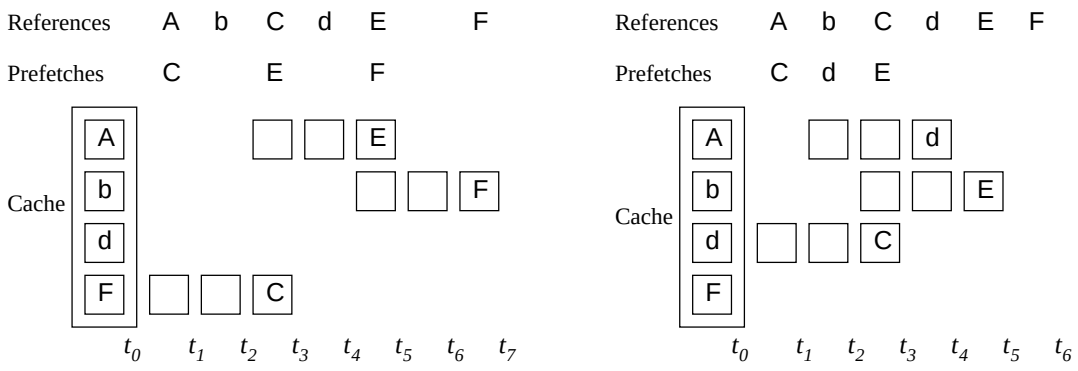
Prefetching has been proposed as a way to exploit the storage parallelism for traditional serial applications [39]. Studies have shown that, for optimal performance, prefetching and caching must be integrated carefully to ensure that they behave synergistically rather than antagonistically. [10, 39].

This paper studies integrated parallel prefetching and caching strategies with multiple disks. The interaction between caching and prefetching is complicated when a system has multiple disks, not only because it is possible to do multiple prefetches in parallel, but also because appropriate cache replacement strategies can alleviate the load imbalance among the disks. The exact nature of the problem depends on the placement of data. Since a disk can serve only one prefetch at a time, a set of blocks can be prefetched in parallel only if they reside on different disks.

Figure 1 gives an example. In figure 1, 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. It takes two time units to fetch a block from disk; each disk can do 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 in the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d, and F. The straightforward approach is to prefetch aggressively 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. This method requires 7 time units.

Figure 1(b) shows another policy 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 (the one holding A,C,E, and F) to the otherwise idle disk (the one holding b and d. This change allows two fetches to proceed in parallel later, thus saving one



(a) An natual 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. Cache size $k = 4$ and fetch time $F = 2$.

time unit.

The example shows that the parallelism of multiple disks can be affected by cache replacement and data placement policies. This is the factor that makes the multiple-disk problem challenging.

We consider two basic algorithms for the d-disk integrated prefetching and caching problem:

- *Regular Aggressive*, the natural extension to d-disks of the near-optimal single-disk algorithm presented in [10], and
- *Reverse aggressive*, a policy which attempts to balance the load on the disks while trying to minimize stall time.

The first of these algorithms is placement-oblivious (prefetching and replacement decisions do not depend on the mapping between blocks and disks) whereas the second is placement-sensitive.

We present theoretical bounds on their performance compared to the optimal offline algorithm (section 3). These bounds apply for any placement of the data on the disks, and indicate that *reverse aggressive* is closer to optimal than *regular aggressive*. We also discuss the effects of data placement on the theoretical results, most notably when the blocks are: *replicated*, in which all files are replicated on all disks, *striped*, in which successive blocks of each file are placed round-robin onto the disks, with the starting point chosen randomly for each file, and *random*, in which each block is placed on a random disk.

We have evaluated the performance of these algorithms on a collection of file-system traces, and across a range of data placement alternatives, including striped, random, etc. The results show that *reverse aggressive* performs well across the data placement schemes and achieves almost linear speedup with balanced data distribution. The performance of *aggressive* is also close to optimal when the placement strategy distributes the file references evenly among the disks (for example, the striping layout). We also evaluated "online" versions of the algorithms that utilize the available lookahead (adavance knowledge) into the file access sequence. The results are in section 4.

2 Integrated Strategies

This section first describes the model and terminology used in the rest of the paper, presents rules that integrated parallel prefetching and caching algorithms must follow, and then proposes two off-line algorithms and four on-line algorithms.

2.1 Model and Assumptions

Our model considers two levels of memory hierarchy: a cache of k data blocks and d (disk) storage devices. The execution of a program makes a known sequence, called *reference sequence*, or *request sequence*, of references $r_1, r_2, \ldots r_n$ to m data blocks. We assume that $k < m < n$ since the solution is trivial if any of these inequalities does not hold.

If a reference hits in the cache, it takes one time unit. Otherwise, the missed block must be fetched from a storage device. The system can either fetch a block on a miss (demand driven) or fetch the block before it is referenced in anticipation of a miss (prefetch). Either case takes F time units. If the cache is full, a cache block must be *evicted* before the fetch is issued to make room for the requested data block. While the fetch is in progress, neither the incoming block nor the discarded block is available for access.

In most of what follows, we will assume that each block resides color the disks for convenience. We let each disk have a unique *color* and refer to the blocks on a disk by their color. Thus d prefetches can be executed concurrently (one per disk), if the blocks fetched are of different colors.

When the program tries to access a block that is not 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 block is prefetched i time units before the reference. The measure of performance is the elapsed time required to serve the entire request sequence, in other words, the number of references plus the total stall time.

The goal of an integrated strategy is to decide together

- when to fetch a block from a disk,
- which disk to fetch from,
- which block to fetch, and
- which cache block to evict when the cache is full

such that the stall time is minimized.

The model is defined to capture the characteristics of a file system with multiple disks while making enough simplifications to conduct theoretical analysis on prefetching and caching. The model considers only read accesses; thus evicting a cache block does not need to write the block back to disk or disks. The time unit models the 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 is the same, and that all disk accesses take the same amount of time. The model also simplifies the real system, where only one direct memory access (DMA) transaction is allowed at a time on the I/O bus, by assuming that fetches on different disks can be performed entirely in parallel.

2.2 Rules for Integrated Strategies

There are four rules that any optimal parallel prefetching strategy must follow.

- *Color-specific optimal fetching*: among all missing blocks of the same color, always fetch the first missing one (the one that will be referenced soonest);
- *Color-specific optimal replacement*: among all blocks of the same color, always replace the one whose next reference is is furthest in the future;
- *Color-specific do no harm*: never evict block A to fetch block B when A's next reference is before B's next reference, and A and B are of the same color;
- *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-replace operations that does not follow these rules can be transformed into one that does, with performance at least as good.

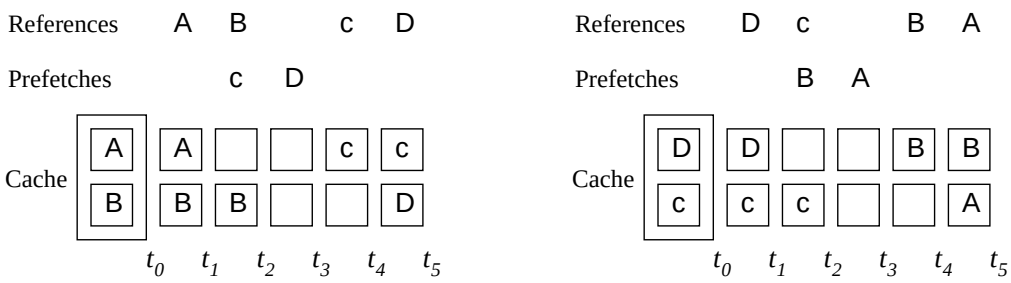
The special cases of these rules for a system with a single disk were proposed in an earlier single-disk study [10]. In the single disk case, these rules specify what to fetch and what to evict, leaving only the decision of when to prefetch. Unfortunately, in the multiple-disk case, these rules only specify what to do among the blocks on the same disk — they do not specify how to deal with the interaction between the disks.

For example, consider just two disks, where we color the blocks on one of the disks red and the blocks on the other disk blue. If there are many red blocks missing in the sequence, say, it may be that the best overall choice for replacement 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. The color-specific rules do not specify which color block to replace.

In fact, the optimal strategy may even replace a blue block to prefetch a a red block though the blue block is referenced before the red block. Again, this is because the loads on the two disks can be imbalanced. If there are many red blocks missing from the sequence, say, but few 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. The blue block may still be fetched in time for its references because the blue disk is lightly loaded.

Using the reverse sequence

An interesting twist allows us to convert multiple-disk prefetching to a more constrained, and hence easier to solve, problem. We begin with an example illustrated in Figure 2: Consider the request sequence (A, B, c, D). Suppose the disk holding blocks A, B, and D is red and the disk holding block c is blue. We will also assume that the cache has two blocks. At time t_0, the first two blocks in the sequence, A and B, reside in the cache. The request to A is served in the unit of time between time t_0 and time t_1. At time t_1, a fetch is initiated to bring c into the cache from the blue disk, evicting A. The request to B is served concurrently, completing at



(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. Cache size $k = 2$ and fetch time $F = 2$.

time t_2. At time t_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 t_3 at which the fetch completes. Between times t_3 and t_4, the request to c is served, and the fetch of D from the red disk completes. Finally, during the unit between times t_4 and t_5, the request for D is served.

Now consider the reverse sequence, and assume that the cache initially contains c and D, the last two blocks in the forward sequence and the first two blocks in the reverse sequence. If we consider each fetch in the forward direction as an eviction in the reverse direction, we obtain a prefetching schedule for the reverse sequence that satisfies the following constraint. *Whereas in the forward direction, we could fetch at most one block of each color at a time, in the reverse direction, we are allowed to evict at most one block of each color at a time.* In our example, in the schedule for the reverse sequence, the request to D is served between times t_0 and t_1. At time t_1, D is evicted in order to start fetching B. The request to c is served concurrently with the fetch of B, completing at time t_2. Since c is blue and D is red, a fetch of A (evicting c) can be started at time t_2. The request for B stalls until time t_3; it is served between times t_3 and t_4. At time t_4, the fetch of A completes, and the request for A is served between times t_4 and t_5.

Obviously, the reversal of fetches and evictions we saw in this example works in both directions. Given a prefetching schedule for the reverse sequence that replaces at most one block of each color, we can construct a prefetching schedule for the forward sequence that fetches at most one block of each color at a time. And of course the elapsed time of both schedules is the same.[1]

This suggests the idea of deriving a near-optimal schedule for the forward direction from a schedule serving the reverse direction that is constrained to replace at most one block of each color at a time. Oddly enough, this approach seems to make the problem easier. The reason 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 replace is the one not needed for the longest time whose color matches the color of the free disk. 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 *color-specific optimal fetching* rule described earlier). In the reverse direction, it is the one needed soonest, period.

2.3 Algorithms

We consider algorithms both when the entire access sequence is known at the beginning (i.e. the offline setting) and when only a limited subsequence of the access string is known at a time (i.e. the limited lookahead online setting).

2.3.1 Offline Algorithms

We propose two algorithms based on the rules for the multiple-disk prefetching problem.

The *aggressive* algorithm

Aggressive is the prefetching algorithm that performs aggressive prefetching in the forward direction. Whenever a disk is free, it prefetches the first missing block of that disk's color, replacing the block whose next reference is furthest in the future among all cached blocks. However, it only starts a fetch if the next access to the replacement is after that to the block being fetched.

[1] This assumes that in both directions the cache initially contains the first k distinct requests in the sequence. In practice, we require *reverse aggressive* to flush all blocks from its cache; this corresponds to starting with an empty cache in the forward direction.

The *reverse aggressive* algorithm

Reverse aggressive is the prefetching algorithm that performs aggressive prefetching (i.e. initiate a prefetch whenever the best replacement choice—the block in the cache that is not used for the longest time among those whose color matches the color of the idle disk— is not referenced until after the first missing block) on the reverse of its input sequence, then derives a schedule to serve the forward sequence as described in section 2.2.

2.3.2 Online Algorithms

Our lookahead model is the following. We assume that there is some l such that at the beginning of the sequence the first l references are known; then at the end of the first l references, the next l references are known, and so on. Many applications can provide such lookahead information [39, 40].

We consider four variations on the previous algorithms. The first two of them assumes that only the lookahead information is available. The second two assume the availability of optimal replacement information in addition to the lookahead information. Many applications can use the knowledge of their file access pattern to provide optimal replacement policies [9].

Online Aggressive

Online aggressive is simply the aggressive algorithm based on that portion of the sequence available. For replacement, it uses whatever information is given by the subsequence; among all blocks with no additional references in the subsequence it uses LRU.

Online Reverse Aggressive

Suppose that at the beginning of the subsequence the set S of blocks are in the cache. *Online reverse aggressive* considers the reverse of the subsequence, and processes it assuming that it starts with the first k requests in the reverse subsequence in the cache (i.e. the last k requests in the forward subsequence), and ends with requests to S. (This reverse prefetching sequence is translated into a forward prefetching sequence in the manner described in section 2.2.)

Online Aggressive With Optimal Replacement Information

This algorithm is the same as *online aggressive* except that it always performs the optimal block replacement (i.e. replaces the block whose next reference is furthest in the future.)

Online Reverse Aggressive With Optimal Replacement Information

This algorithm figures out the forward prefetching schedule exactly as *online reverse aggressive* does. It then modifies the forward prefetching schedule using the optimal replacement information, i.e. whenever the schedule calls for replacing of a block of some color, it replaces the furthest referenced block of the same color, and whenever the schedule calls for prefetching of a block of some color, it prefetches the next missing one of that color.

3 Theoretical analysis

3.1 Arbitrary Layout: Summary

Our theoretical analysis assumes that we are given a fixed placement of blocks on the d disks. For any given fixed layout, we attempt to come as close as possible to the optimal prefetching schedule for that placement, i.e. the schedule which minimizes the elapsed time. All of the upper bounds we present below hold *for any choice of data placement*, be it striped, random, or chosen by an adversary. In section 3.2, we discuss the impact of data placement on these results.

Our main results are the following: On any reference string R, and for $d \geq 2$ disks, the elapsed time of *aggressive* is at most d times the elapsed time of the optimal prefetching strategy on R. We can also show that the bound on *aggressive* is nearly tight for $d = 2$: 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). Due to space limitation the analysis is described in Appendix A.

We are also in the process of writing up a formal proof that: On any reference string R, and for $d = 2$ disks, the elapsed time of *reverse aggressive* is at most $(1 + 2F/k)$ times the elapsed time of the optimal prefetching strategy on R. Although the formal proof is out of the scope of this paper, a detailed outline of the ideas involved can be found in Appendix A.

3.2 Data Placement

We have seen evidence that, at least for two disks, there is not much room for improvement in the performance of *reverse aggressive*. In contrast, we saw that in the worst case *aggressive*'s performance on two or more disks is at least a factor of two worse than optimal.

There are two fundamental reasons for these differences between *aggressive* and *reverse aggressive*:

- The replacement decisions made by *aggressive* are "color-blind". It chooses replacements to make without consideration of the load on the disks. As a consequence, the replacement decisions it makes 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. In contrast, since *reverse aggressive* greedily replaces from as many disks as possible on the reverse sequence, it is automatically ensuring close to the highest degree of parallelism possible on fetches in the forward sequence.

- *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 are wasted, since those blocks may subsequently need to be replaced before they are requested, so that the result can be that there are more cache misses overall. In contrast, since *reverse aggressive* is greedy in the reverse direction, it is fetching pages in the forward direction just in time (to the extent possible) for them to be used, hence performing close to the best replacements possible for those fetches. [2]

These differences between *aggressive* and *reverse aggressive* largely disappear, however, when the placement of the data is more balanced. For example, *aggressive*'s factor of 2 performance ratio for 2 disks arises when the holes *aggressive* is filling tend not to be alternately red and blue. Indeed, an examination of the proof for *aggressive* indicates that the actual performance ratio between *aggressive* and Opt is equal to the average length of a *2-fetch*. Roughly speaking, the length of a 2-fetch is a measure of how long it takes *aggressive* to fetch its first two missing blocks. If these two blocks are of different colors, then the length of the 2-fetch is F, since *aggressive* can fill them in parallel. On the other hand,if they are of the same color, the average length of a 2-fetch can be as large as 2F. This observation generalizes to any number of disks: the more the faults alternate between the d disks, the more parallelism *agressive* can achieve and hence the closer its performance is to optimal.

Therefore, we'd expect that:

- If the requests in the sequence are heavily biased towards a small subset of the disks, in a non-uniform fashion throughout the request sequence, *reverse aggressive* should outperform *aggressive*, since it pays attention to the load on the disks, and does not prefetch too greedily.

- If layout of data is random across the d disks, *aggressive* should do somewhat better, since then at least on average, the distribution of requests to each disk is uniform.

- If layout of data is striped, and we are looking at file system accesses which contain a lot of sequential access, then we'd expect the request sequence to be almost exactly round-robin between the disks, which should allow *aggressive* to perform close to optimally as well.

As the simulation results will show later, these observations are indeed true in practice.

3.2.1 Replication

The ultimate in perfect layout is full replication, where each block is resident on every disk. The following is a straightforward extension of the 1 disk theorem for Aggressive from [10].

Theorem 1 *For any reference string R, the elapsed time of Aggressive on d disks with full replication is at most $(1 + F/k)$ times the elapsed time of the optimal algorithm on d disks with full replication.*

4 Simulation results

We used trace-driven simulation to evaluate the performance of the *reverse-aggressive* and *aggressive* algorithms under varying assumptions about fetch cost and data placement. We also simulated the behavior of online versions of the two algorithms, in the case where only limited information is available about the future reference stream.

4.1 Simulation framework

Our trace-driven simulator uses a machine model that corresponds to the theoretical model: each reference takes one time unit, each fetch takes exactly F time units, and the cache holds k blocks. The reference stream is taken from a trace of real file behavior.

File Access Traces

We used two sets of traces. The first set is from the Sprite file system traces from UC Berkeley [36]. The Sprite traces consist of five sets, recording about 40 clients' file activities over a period of 48 hours (traces 1,

[2] Note the difference between this and prefetching in the forward direction, trying to fetch each missing block F units ahead of the reference to it. With this kind of strategy, future faults that will be generated are not taken into account, and could cause unanticipated stalling later in the processing of the sequence. By considering the reverse sequence, future faults in the forward sequence are taken into account in generating the prefetching schedule.

2 and 3) or 24 hours (traces 4 and 5). For each client we extracted its file activities to Sprite's main file server. From these client traces we eliminated those 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 (7MB is the average file cache size for the Sprite clients). The result is 17 client traces[3]. We denote these traces as Sprite[1-17].

The Sprite traces are not read-dominated: about 10% to 30% of the accesses are writes. However, we treat write accesses as if they are read accesses in our simulations. Though inaccurate, the simulation can still produce some insights on how the algorithms might perform in practice.

The second set of traces was collected by tracing several applications running on a DEC 5000/200 workstation under Ultrix 4.3. This set of traces have the property that the running time of the applications are dominated by disk read accesses. The applications are:

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 [33], 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 [4]). Since the result relation is small, most of the file accesses are reads.

postgres-selection: the Postgres relational database system executing a selection query of choosing 2% of the tuples from an indexed 32MB relation. The selection query is part of the Wisconsin Benchmark suite [4] and uses the indexed search.

xds: a 3-D data visualization program, XDataSlice, generating animations of planar slice images of a 220x220x220 data volume. The animations are along X, Y and Z axes, with a stride of 10 (i.e. 22 images in each animation).

Below we usually report running times averaged over the traces. We use a weighted average, where the weights are such that each set of traces has half of the total weight; the weights are equal within each set.

[3]client 29, 33, 53, and 81 of trace 1, client 60, 62, 75, 82 of trace 2, client 77, 53, 63 and 18 of trace 3, client 9, 77, 56, 18 of trace 4, and client 18 of trace 5

Data Placement

We consider several schemes for placing data onto disks:

- striping: blocks in a file are distributed round-robin over the disks, with the starting point chosen randomly for each file;
- random: every file block is allocated on a randomly-chosen disk;
- biased-striping: same as striping, except that half of the disks are considered "hot", and the first block of a file is allocated on a hot disk 70% of the time;
- by-File: all blocks of a file placed on the same (randomly chosen) disk (modeling a system where each disk is a separate file system partition);
- Total-Replication: every file block is replicated on every disk. (Although this scheme is not practical, we use it to provide an upper bound on the speedup achievable from parallel disks.)

Of the four practical schemes, striping and random balance the disks' load best. Biased-striping tends to concentrate references to small files (which occupy only one block) on the hot disks, but the load from large files is well distributed. By-file can lead to severe short-term load imbalance, as a sequence of consecutive accesses to a file all require the same disk.

4.2 Results

We now present the simulation results. For each configuration, we compute two metrics: time per reference and normalized execution time. Time per reference is simply the ratio of the running time to the number of references in the trace; if there are no stalls then time per reference will be 1. For each trace, time per reference is proportional to running time, so this metric is useful when comparing running time as parameters change.

Normalized execution time is the running time in a configuration, divided by a lower bound on the running time for that configuration. Since the lower bound changes with the configuration, this metric is best used for comparing different policies under a single configuration. To get the lower bound, we compute the running time of aggressive under the total-replication placement strategy, and then divide by $(1 + F/k)$. By theorem 1,

this lower bound is valid, and of course could still be weak since theorem 1 reflects the worst case.

For those readers interested in the full data, results for time per reference are found in appendix B. These data clearly show near-linear speedup as the number of disks is increased, in almost all cases.

4.2.1 Offline Algorithms

In this section we present simulation results assuming the complete sequence of file references is available beforehand. In other words, the algorithms have perfect knowledge of the access sequence.

We first compare the performance of *regular aggressive* and *reverse aggressive*. For both traces, we simulated the algorithms with $F = 20$ and $F = 200$, varying the number of disks. The cache size was set to be 7MB (or $k = 896$ blocks of 8 kbytes each) for the Sprite traces, and 6.4MB ($k = 820$ blocks) for the Ultrix traces.

Striping

Results under striping are shown in Figure 3. *Reverse aggressive* is usually close to optimal: in fact, in many cases its performance is nearly identical to that of regular aggressive under total replication, which means that its performance is within a factor of $1 + F/k$ of optimal. (This is especially true for the case of $F = 20$, although the difference between reverse aggressive and the total replication case becomes bigger when $F = 200$.) An interesting implication is that total replication offers only a slight advantage over striping for these kinds of traces. Though total replication allows maximum flexibility for prefetching in parallel, reverse aggressive can take advantage of all the available parallelism under the striping distribution.

On average, *regular aggressive* performs close to *reverse aggressive*. The differences are quite small for the $F = 20$ case. However, as the fetch time increases, the advantage of *reverse aggressive* becomes clearer, since balancing the loads between the disks becomes more important.

We also experimented with doubling the cache size for these traces. The results (which are not shown here) are very similar to the ones shown here except that the differences between the algorithms are smaller. The reason is that there are fewer cache misses, so the elapsed time is less affected by the disk accesses.

In summary, both *regular aggressive* and *reverse aggressive* perform well for the striping distribution and achieve near-linear speedup as the number of disks is increased.

Random

Figure 4 shows the normalized execution time with random data placement. We see that *reverse aggressive* is still very close to optimal.

However, *regular aggressive* performs much worse in some cases, especially when the fetch time is large ($F = 200$). In one case the elapsed time of *regular aggressive* is more than triple the optimal value. These results illustrate that *reverse aggressive* is a much more robust algorithm than *regular aggressive*, even when the distribution is fairly well balanced, as it is under a random distribution.

Biased-Striping

Figure 5 shows the results for the biased-striping distribution scheme. The results are more or less between those of the striping and random distribution schemes.

Though biased-striping favors the first half of the disks for the starting block of a file, it allocates blocks consecutively on disks within a file. Thus, for applications that primarily use large files, it still balances the load on the disks well. Therefore, on average *reverse aggressive* still achieves near-linear speedups.

By-File

Figure 6 shows results under the by-file distribution, in which all blocks of a file reside on one disk. Unless the application uses mostly small files, this distribution can create serious load imbalance among the disks. The figure shows that there is a clear difference between the performance of *regular aggressive* and *reverse aggressive*. *Reverse aggressive* apparently deals with this imbalance much better.

Summary of Results for Offline Algorithms

Overall, we see that the distribution of file data on disks is critical for the performance of parallel prefetching. Under striping, which distributes the load evenly, the speedup from parallel prefetching on D disks can be very close to D, and *regular aggressive* can perform almost as well as *reverse aggressive*. However, when the load is not balanced, as in the case of by-file and biased-striping, the advantage of *reverse aggressive* becomes clear. In an unbalanced system, *reverse aggressive* can perform significantly better than *regular aggressive*. *Reverse-aggressive* is relatively insensitive to load imbalance.

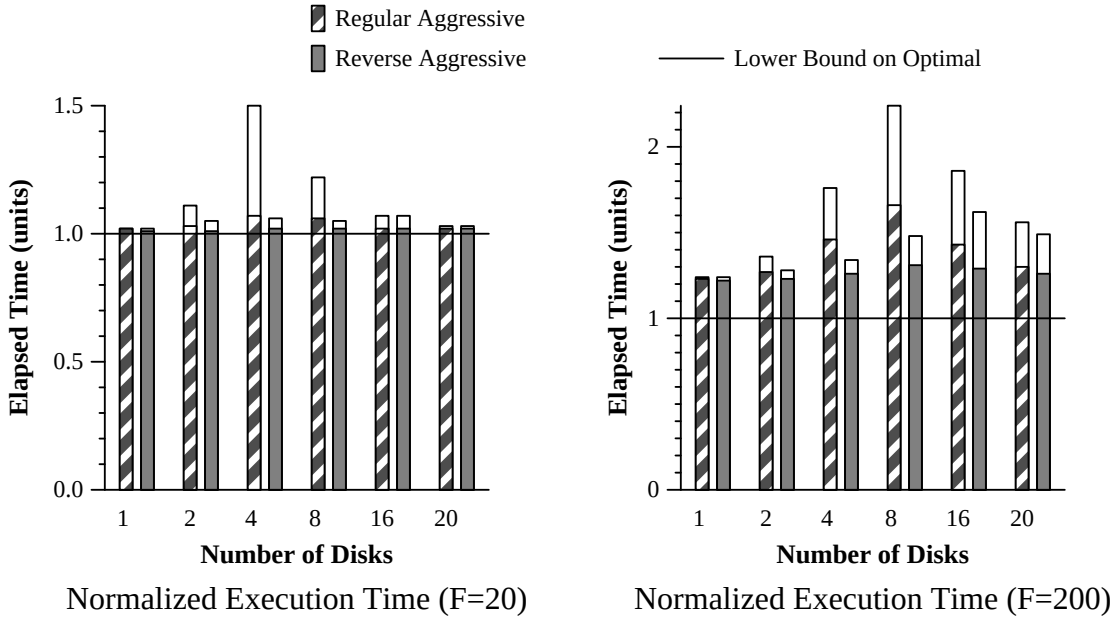


Figure 3: Normalized execution time under the striping distribution. The shaded part of each bar represents the average value, and the overall height represents the worst case. The cache size for the sprite traces is 7MB (896 blocks) and for the ultrix traces 6.4MB (820 blocks).

4.2.2 Study of Online Algorithms

In this section we report simulation results on the online algorithms. We simulated four online algorithms (described in section 2.2) on the traces with the striping distribution, under $F = 20$. We used a lookahead of 750 references. The average normalized execution time is shown in figure 7.

The results show that the online versions of *regular aggressive* and *reverse aggressive* perform about 10% to 50% worse than the offline algorithms. Of course, this depends on the lookahead distance. In addition, the online versions of the two algorithms perform about the same.

One significant factor is the availability of optimal replacement information. Without optimal replacement, the online algorithms perform 25% to 35% worse than the offline algorithms in our simulations. This shows that even in the presence of lookahead into file references, good application-specific replacement information (i.e. hints about file access patterns) is still important.

Combined with our results in the offline case, we think that *reverse aggressive* with optimal replacement (if possible) remains the algorithm of choice in the more practical online case.

5 Related work

Caching and prefetching have been known techniques to improve storage hierarchies for many years [47, 2, 19]. In architectures, the work on caching and prefetching has been focussed on bridging the performance gap between CPU and main memory [44]. Research using caching and prefetching in database systems [45, 13, 37, 15] showed that it is important to use applications' knowledge to perform caching and prefetching.

File caching and prefetching have become standard techniques for sequential file systems [19, 32, 25, 36, 46, 8, 23, 11, 40]. The most common prefetching approach is to perform sequential readahead [19, 32, 34]. The limitation of this approach is that it only benefits applications that make sequential references to large files. Another large body of work has been on predicting future access patterns [18, 46, 37, 15, 23]. Unfortunately, few of them considered the interaction between caching and parallel prefetching as discussed here.

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 and to tolerate failures [28, 43, 3, 14, 12, 38, 22, 30, 35, 21, 5]. These techniques were used in designing hardware disk array systems [7, 27] and parallel or distributed file systems. The Intel CFS [41] and PFS [26] allow users to stripe within a file to multiple disks over their multicomputer network to achieve high bandwidth. The research prototype

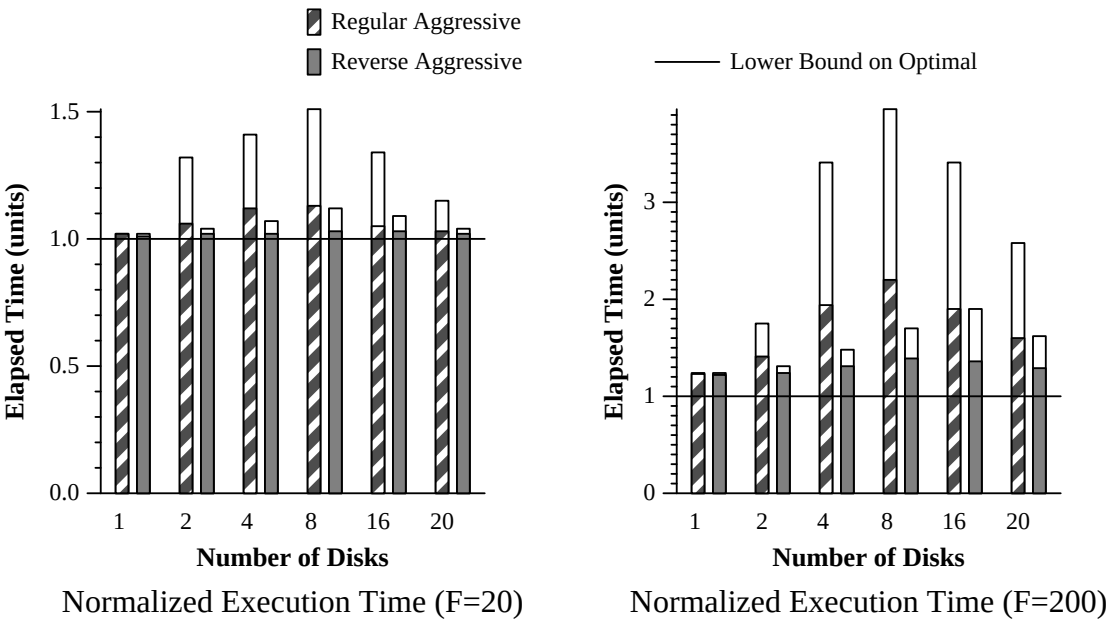


Figure 4: Average normalized elapsed time with fetch time 20 and 200, under the random distribution. The shaded part of each bar represents the average value, and the overall height represents the worst case.

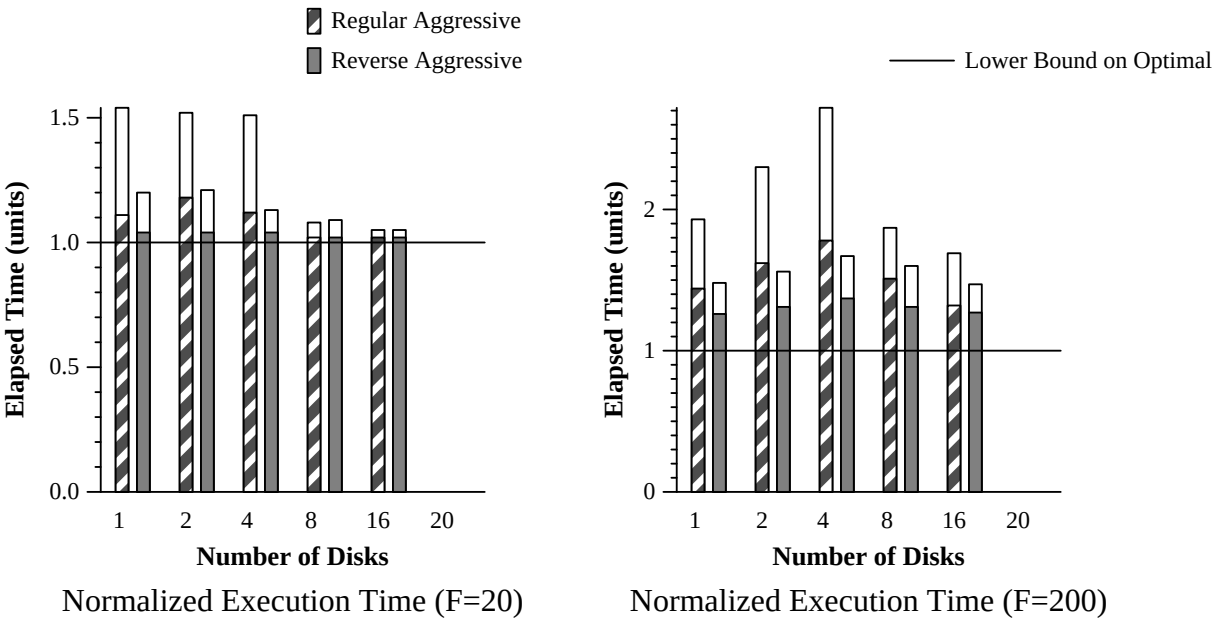


Figure 5: Average normalized execution time with fetch time 20 and 200, under the biased-striping distribution. The shaded part of each bar represents the average value, and the overall height represents the worst case.

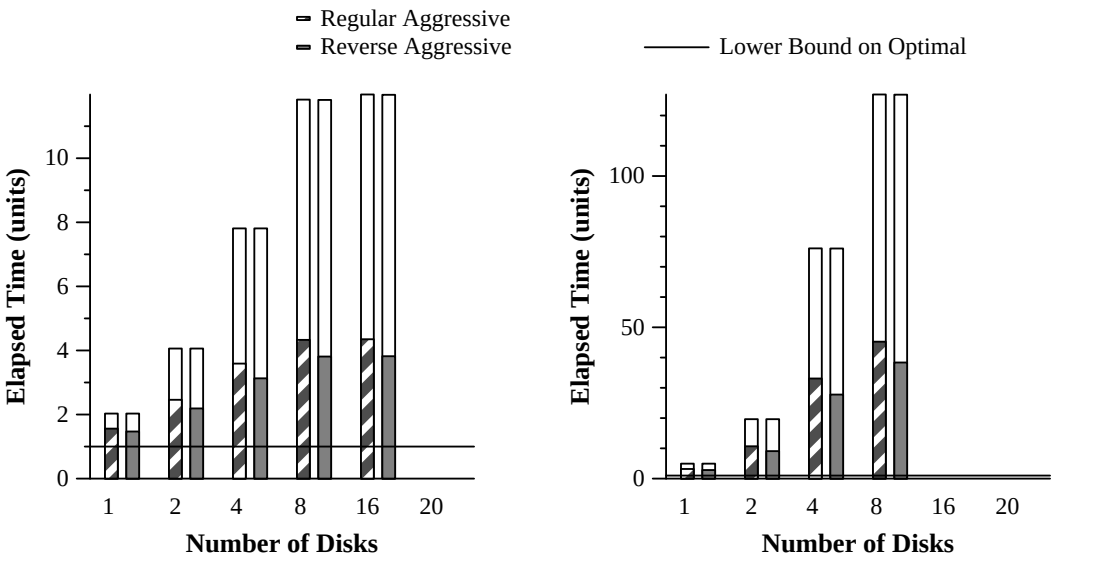


Figure 6: Average normalized execution time with fetch time 20 and 200, under the byfile distribution. The shaded part of each bar represents the average value, and the overall height represents the worst case.

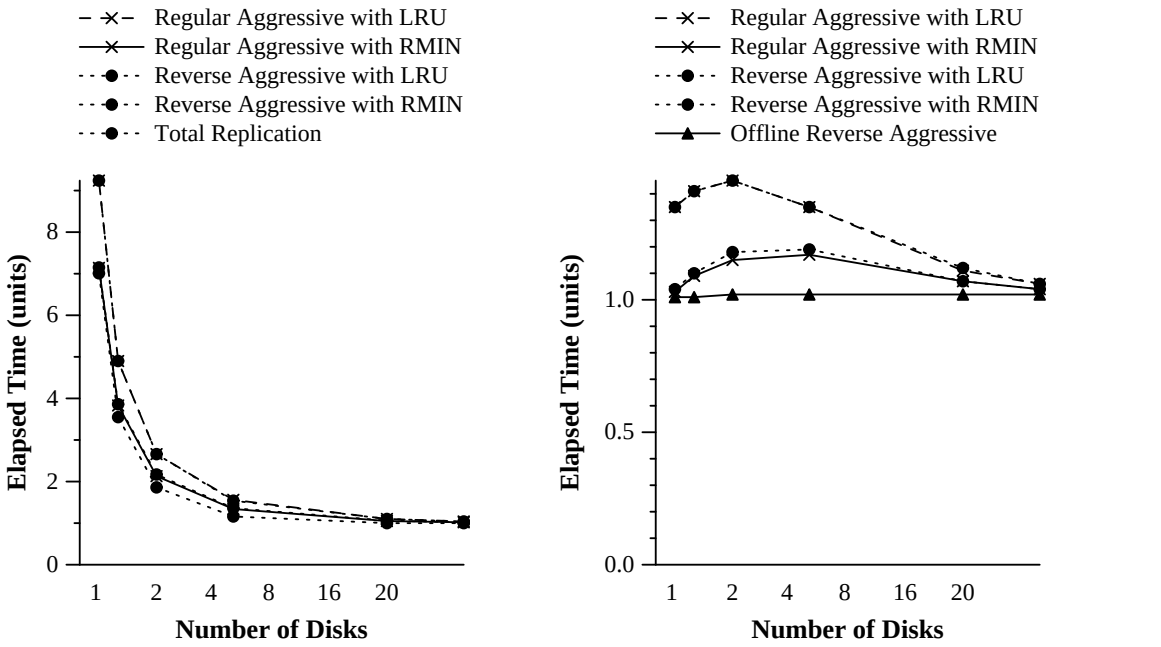


Figure 7: The average of the normalized elapsed time under the fetch time of 20, with a online lookahead of 1500.

file systems such as Bridge [16], Swift [6], Vesta [20], LFS on RAID [17], and Zebra [24] allow various striping strategies to provide parallelism. Also, there are techniques [42] to take advantage of redundant disks to optimize disk arm scheduling. Recently, caching and prefetching have also been studied for parallel file systems [18, 29, 31, 39, 48].

Although much work has been done in file caching and prefetching, little study has been done on how to integrate caching and prefetching together. Recent studies showed [10, 9] that it is important to integrate prefetching, caching and disk scheduling together and that a properly integrated strategy can perform much better than a naive strategy. Patterson et al [40] used a model to control buffer allocations for caching with prefetching as a way to integrate caching and prefetching for both the single-disk case and the parallel-disk case. Their work considered accomodating both prefetching and non-prefetching applications in a multi-process environment, while ours concentrates on minimizing the elapsed time of applications in single-process workloads.

It is difficult to compare our approach to existing approaches. Unlike the single-disk case, there do not exist "standard" parallel prefetching algorithms. Other approaches include the one stripe lookahead prefetching on RAID arrays, and Patterson et al [40]'s prefetch-horizon algorithm. The one stripe lookahead only benefits applications that use large files, and would perform little prefetching for other applications [40]. The prefetch-horizon algorithm is aimed at optimizing both individual prefetching applications' performance and the performance of the whole system in a multi-process environment, and thus is not concerned with single application performance alone.

For these reasons, the baseline against which we compared our algorithms was a lower bound on the optimal performance.

6 Conclusions

This paper studied how to integrate file caching and prefetching for multiple disks. It presented two algorithms, *aggressive* and *reverse aggressive*, and evaluated them via both theoretical analysis (for worst case performance) and trace-driven simulations (for average case performance).

The theoretical analysis shows that in the worst case, *aggressive* can perform at least twice as poorly as optimal. The idea of reversing the sequence to transform the problem to a easier one led to the discovery of the new "reverse aggressive" algorithm which is close to optimal. The key features of *reverse aggressive* are that it balances the load on multiple disks and keeps the disks in pace with each other.

Using trace-driven simulation, we showed that *reverse agressive* does perform better than aggressive in practice. Reverse aggressive has a particularly large advantage when the data blocks are not distributed evenly between the disks. Both algorithms were able to improve performance as the number of disks increased, by exploiting opportunities for prefetching in parallel. Perhaps surprisingly, replicating data across the disks, even if taken to the extreme of full replication, offers no significant performance advantage over simple striping, if a good prefetching and replacement algorithm is used. Finally, we found that limited-lookahead versions of reverse-aggressive and aggressive perform well, especially if replacement decisions are made wisely.

This study has several limitations. First, most of our results assume full knowledge of the request stream is available in advance. Although this is in general not realistic, there are some applications that have this sort of knowledge. Better understanding of limited-lookahead versions of the algorithms is one area of future work. Second, we consider only read traffic, ignoring writes. Writes make the problem considerably harder, but clearly the write problem must be faced eventually. Finally, our simulations do not model disk behavior in detail. Although this may cause us to miss some effects that would arise in practice, it has the virtue of making the simulation correspond more closely to the theoretical model.

References

[1] Anonymous. Submission to 1996 ACM Symposium on Theory of Computing. In preparation.

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

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

[4] Jim Gray. *The Benchmark Handbook*. Morgan-Kaufman, San Mateo, CA. 1991.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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. To appear.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

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

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

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

[46] 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.

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

[48] 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 A: Theoretical Results on Arbitrary Layout

Terminology

We follow the lead of [10], so that in what follows. when we talk about prefetching algorithms for a given reference sequence, we imagine that there is a *cursor* which advances at a fixed rate along the reference string until it hits an access to a file block that is not present in the cache. At that point, the cursor stalls until that block arrives in the cache.

At any point in processing the sequence, a *hole* is a block that is not present in the cache. At a given cursor position, the i-th hole is the i-th distinct block requested after that cursor position that is *not* present in the cache at that time.

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).

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 $b4$, 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 $k = F + 2$ for simplicity. It is easily generalized for 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).

Aggressive: Upper Bound

We present in some detail only the upper bound for *aggressive* for $d = 2$ disks.

Theorem 2 *On any reference string R, the elapsed time of* aggressive *with two disks on R is at most twice the elapsed time of the optimal prefetching strategy on R.*

Proof: We begin by describing the idea of the proof. We will be using the concept of *domination* from [10]. We say that algorithm A's state at time t *dominates* algorithm B's state at time t' if the following two conditions hold:

- A's cursor position at time t is greater than or equal to B's cursor position at time t'.
- For each i, A's i-th hole is at a position greater than or equal to B's i-th hole.

In the analysis of aggressive prefetching with one disk, it was shown that if A's state dominates B's state, and each algorithm initiates a fetch, A will continue to dominate B when the fetch is completed. This result was referred to as the *domination lemma*[10].

In order to apply this lemma to two disks, we must be sure that when we are comparing a fetch A initiates to a fetch B initiates that in both cases the hole being filled is *the first missing hole*. If not, the domination lemma does not hold.

In general, we can not ensure that two parallel prefetches *aggressive* initiates will fill the first two holes, since both of these holes may be of the same color. However, we do know that by the time *aggressive* completes two prefetches on the same disk, the first two holes that were present (and perhaps others) have been filled.

Therefore, our proof strategy is to run Opt at half-speed, so that during each subsequence of time in which *aggressive* fills *at least* its first two holes, Opt can fill *at most* its first two holes. We will be able to show inductively that at the beginning of each of these subsequences, *aggressive*'s state dominates Opt's state, and therefore the overall running time of *aggressive* is at most twice that of Opt's.

More precisely, we will need the concept of a *2-fetch*. The first 2-fetch begins at the time that *aggressive* initiates its first fetch, say at time t_0. In general, suppose that the i-th 2-fetch begins at time t_i. At time t_i, *aggressive* has a certain set of holes, say $h_1^i, h_2^i, h_3^i, \ldots, ...$, where h_1^i is the first, h_2^i is the second and so on. Let j be the lowest numbered hole which *aggressive* starts fetching strictly after time $t_i + F$, and let t_{i+1} be the time at which *aggressive* initiates that fetch. Note that j is at least 3.

We prove the following inductively:

Inductive Hypothesis:

- Let t_i be the ith 2-fetch that is initiated, and suppose that at time t_i *aggressive*'s state dominates Opt's state at time t'_i. Then at time $min(t_{i+1}, t_i + c_i)$, *aggressive*'s state dominates Opt's state at time $t'_{i+1} = t'_i + F$, where c_i is the time at which the fetches for h_1^i and h_2^i complete.
- If at some time t, *aggressive* is not in the middle of a 2-fetch, and *aggressive*'s state at time t dominates Opt's state at time t', then *aggressive*'s state at time $t + 1$ dominates Opt's state at time $t' + 1$.

Proof of Inductive Hypothesis: The proof of this is similar in spirit to the proof of the domination lemma in [10]. Part 2 is trivial, since if *aggressive* is not in the middle of a 2-fetch, then *aggressive*'s state is "optimal": it has in its cache the next k distinct blocks to be requested. In order to show the first part, we consider the one or two fetches that Opt *completes* during the period $(t'_i, t'_{i+1} = t_i + F]$. Since *aggressive*'s state at time t_i dominates Opt's state at time t'_i, both fetches which *aggressive* begins during the ith 2-fetch are initiated in a better state than the corresponding fetches Opt completes during the period $(t'_i, t'_{i+1}]$. Therefore *aggressive*'s replacement opportunities on those two fetches are at least as good as Opt's replacement opportunities, and a slightly extended version of the domination lemma shows that *aggressive*'s state at time t_{i+1} dominates Opt's state at time t'_{i+1}. □

This theorem can be extended to d disks in a rather straightforward way and will be presented in detail in [1].

Theorem 3 *On any reference string R, the elapsed time of* aggressive *with d disks on R is at most d times the elapsed time of the optimal prefetching strategy on R.*

Intuition for *reverse aggressive*'s near-optimality

In this section, we present an outline of a proof and the intuition for why on any reference string R, the elapsed time of *reverse aggressive* on R is at most $1 + 2F/k$

times the elapsed time of the optimal prefetching strategy on R. The details will be presented in a companion theoretical paper [1].

In what follows, we refer to a red fetch as a fetch that replaces a red page, and a blue fetch as a fetch that replaces a blue page. Recall that since we are looking at the reverse sequence we can only have one of each at any time.

The key lemma we need is a strengthening of the domination lemma described earlier. As previously mentioned for the multi-disk *aggressive* proof, the domination lemma tells us that if algorithms A and B initiate fetches at times t and t' respectively, and A's state at time t dominates B's state at time t', then A's state upon completion of the fetch (time $t + F$) dominates B's state at the completion of its fetch (time $t' + F$).

At the highest level the proof for *reverse aggressive* follows the same structure as the proof of *aggressive* for 1 disk from [10]. Roughly speaking that proof worked as follows. The sequence of requests was divided up into *phases,* maximal subsequences consisting of requests to k distinct blocks. The goal was to maintain the invariant that at if *aggressive* was given an extra F time units to complete each phase, then at the beginning of each pahse, *aggressive*'s state would dominate Opt's state. More specifically, within each phase, *aggressive* was given a one fetch head-start. and then it was shown that each of *aggressive*'s fetches in the phase could be matched to one of Opt's fetches, so that inductively at the ends of the corresponding fetches *aggressive*'s state dominated Opt's state. The reason the one fetch head-start was necessary at the beginning of each phase was in order to deal with the fact that the very first fetch *aggressive* does could be premature. I.e. Opt, by waiting a small number of steps, could perform a significantly better replacement and could pass *aggressive* by at the beginning of the next phase. The reason this could only happen once per phase is that every replacement *aggressive* makes within a phase is of a block whose first request is not in the same phase. Indeed, if there are i holes in the current phase, then there are i pages in the cache that are not requested in this phase, and since *aggressive* always throws out the page whose next request is furthest in the future, one of these will be replaced.

We'd like is to use a similar proof outline for 2-disk *reverse aggressive* i.e. to match *reverse aggressive*'s blue fetches up to Opt's blue fetches and *reverse aggressive*'s red fetches up to Opts red fetches. and apply the domination lemma to each. Unfortunately, if Opt replaces a blue page, and *reverse aggressive* a red page, or vice versa, the domination lemma does not hold. Therefore, we are obliged to show that if we match evictions of the same color then a stronger notion of domination is maintained

Even more troubling, however, is that fact that it is no longer the case that *reverse aggressive* never creates new holes within the new phase. Indeed, of the, say i, pages in the cache that are not requested in this phase, it could be that a majority are blue. (In what follows, we will refer to the color with the majority of replacement opportunities in the phase as the busy disk. In this discussion, it will always be the blue disk.) This would mean that all the blue fetches could be performed without creating new blue holes within the phase. The red fetches, however, may require creating new red holes within the phase. These new red holes could potentially destroy our ability to keep *reverse aggressive*'s cursor ahead of Opts.

We get around this with a two-step argument which goes roughly as follows:

- We stop Opt for F steps when *reverse aggressive* initiates its first blue fetch for the phase, and then *again* when *reverse aggressive* initiates its first red fetch for the phase. The fact that the busy disk, in this case, the blue disk, is fetching continuously as long as there are holes in the phase implies that during the F steps when *reverse aggressive* initiates its first red fetch for the phase, *reverse aggressive* completes *two* fetches, and thereby gains at least a *one-hole lead* in the phase relative to Opt. We say that *reverse aggressive* has a one-hole lead over Opt, if Opt has at least one more hole remaining in the phase than *reverse aggressive* has, and *reverse aggressive*'s i-th hole dominates Opt's (i+1)st hole in the phase.

- We show that the one-hole lead in the phase is maintained throughout the phase by repeated application of the *Extended First Opportunity* principle. This principle says that if Opt is ever able to issue a prefetch (i.e. has a free disk and some block which can be replaced), and elects not to, then we can assume that every block of that disk's color that is in its cache at that time stays in its cache until it is next requested. Indeed, if not, then that block could have been replaced earlier, and this could only have improved Opt's state.

  The one-hole lead ensures that whenever *reverse aggressive* does a premature red fetch, *reverse aggressive*'s cursor still doesn't fall behind Opt's. Since subsequent red-fetches of Opt's can always be matched, the one-hole lead is regained at the end of each continuous sequence of red fetches. Therefore, *reverse aggressive* makes it to the beginning of the next phase, maintaining domination of holes and domination of cursor position.

mapping	F	Algorithm	Number of Disks					
stripe	F=20		1	2	4	8	16	20
		Regular-Aggressive	7.01	3.63	1.98	1.21	1.01	1.00
		Reverse-Aggressive	7.01	3.57	1.88	1.17	1.01	1.00
		Total-Replication	7.01	3.55	1.86	1.16	1.00	1.00
stripe	F=200		1	4	16	64	128	200
		Regular-Aggressive	69.69	18.27	5.33	1.82	1.21	1.07
		Reverse-Aggressive	69.70	17.67	4.64	1.44	1.09	1.04
		Total-Replication	69.69	17.50	4.47	1.33	1.02	1.00
random	F=20		1	2	4	8	16	20
		Regular-Aggressive	7.01	3.73	2.07	1.31	1.04	1.01
		Reverse-Aggressive	7.01	3.57	1.89	1.19	1.01	1.00
		Total-Replication	7.01	3.55	1.86	1.16	1.00	1.00
random	F=200		1	4	16	64	128	200
		Regular-Aggressive	69.69	20.14	6.91	2.40	1.61	1.31
		Reverse-Aggressive	69.70	17.85	4.82	1.53	1.15	1.07
		Total-Replication	69.69	17.50	4.47	1.33	1.02	1.00
biased	F=20		1	2	4	8	16	20
		Regular-Aggressive	-	3.89	2.16	1.27	1.01	1.00
		Reverse-Aggressive	-	3.63	1.92	1.19	1.01	1.00
		Total-Replication	-	3.55	1.86	1.16	1.00	1.00
biased	F=200		1	4	16	64	128	200
		Regular-Aggressive	-	20.46	5.86	1.94	1.27	1.08
		Reverse-Aggressive	-	18.09	4.79	1.50	1.10	1.04
		Total-Replication	-	17.50	4.47	1.33	1.02	1.00
byfile	F=20		1	2	4	8	16	20
		Regular-Aggressive	-	5.65	4.75	4.39	4.29	4.28
		Reverse-Aggressive	-	5.36	4.25	3.88	3.78	3.75
		Total-Replication	-	3.55	1.86	1.16	1.00	1.00
byfile	F=200		1	4	16	64	128	200
		Regular-Aggressive	-	46.68	41.14	39.47	39.31	-
		Reverse-Aggressive	-	41.76	35.57	33.80	33.59	-
		Total-Replication	-	17.50	4.47	1.33	1.02	-

Appendix B:

The following table contains additional simulation data.