Appears in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (OSDI), Seattle, WA, October 28-31, 1996, pp. 19-34.

A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching

Tracy Kimbrel [*] Andrew Tomkins [†] R. Hugo Patterson [‡] Brian Bershad[*] Pei Cao [§]
Edward W. Felten [¶] Garth A. Gibson[†] Anna R. Karlin[*] Kai Li[¶]

Abstract

High-performance I/O systems depend on prefetching and caching in order to deliver good performance to applications. These two techniques have generally been considered in isolation, even though there are significant interactions between them; a block prefetched too early reduces the effectiveness of the cache, while a block cached too long reduces the effectiveness of prefetching. In this paper we study the effects of several combined prefetching and caching strategies for systems with multiple disks. Using disk-accurate trace-driven simulation, we explore the performance characteristics of each of the algorithms in cases in which applications provide full advance knowledge of accesses using hints. Some of the strategies have been published with theoretical performance bounds, and some are components of systems that have been built. One is a new algorithm that combines the desirable characteristics of the others. We find that when performance is limited by I/O stalls, aggressive prefetching helps to alleviate the problem; that more conservative prefetching is appropriate when significant I/O stalls are not present; and that a single, simple strategy is capable of doing both.

1 Introduction

Recently there has been a great deal of interest in prefetching from parallel disks as a technique for improving the I/O performance of sequential applications. In this paper, we study prefetching and caching strategies for multiple disks in the presence of application-provided knowledge of future accesses. We compare the performance of four algorithms:

1. *Fixed horizon* is simple to implement, and has near-optimal performance when sufficient I/O parallelism is available, but can be suboptimal in I/O-bound situations.

2. *Aggressive* is also simple to implement, is close to optimal for a single disk and for well-laid-out data on multiple disks, but can be suboptimal for multiple disks when the load on the disks is unbalanced.

3. *Reverse aggressive* is substantially more complex, but is provably close to optimal for all configurations in a uniform fetch-time model of disk accesses.

4. *Forestall* is a new algorithm, representing an attempt to combine the desirable characteristics of the other three algorithms.

Using trace-driven simulation on a collection of file access traces, we compare the performance of these algorithms assuming an environment in which a single process is running and full advance knowledge is available.

1.1 Motivation

Our work is motivated by recent advances in technology that have made magnetic disks both cheaper and smaller. As a result, parallel disk arrays have become an attractive means for achieving high performance from storage devices at low cost [15, 28, 24]. Independently accessible multiple disks offer the advantage of both increased bandwidth and reduced contention on individual disk arms. However, many applications do not benefit from this parallelism because their I/O accesses are serial. This problem is particularly severe for read-intensive applications. Write performance is less important as write behind strategies can mask update latency. Read-intensive applications that stall for I/O a significant fraction of their running time include text search, 3D scientific visualization, relational database queries, multimedia servers and object code linkers.

Many of these applications have *predictable access patterns* [25, 1, 18]. The ability to provide the file system with hints about future references has motivated research into the design of policies that use this information to reduce I/O overhead [25, 26, 7, 6]. The two key techniques that are enabled by detailed information about future accesses are deep prefetching and better-than-LRU cache replacement.

This paper explores the tradeoff between aggressive prefetching and optimal cache replacement. The decision to

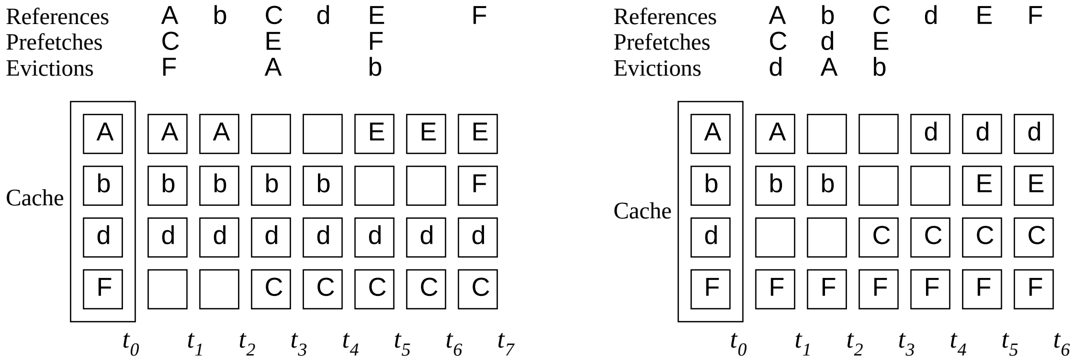
[*] Dept. of Computer Science and Engr., University of Washington, Seattle WA ({tracyk,bershad,karlin}@cs.washington.edu)

[†] School of Computer Science, Carnegie Mellon University, Pittsburgh PA ({andrewt, garth}@cs.cmu.edu)

[‡] Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA (rhp@cs.cmu.edu)

[§] Computer Sciences Dept., University of Wisconsin - Madison, Madison WI (cao@cs.wisc.edu)

[¶] Dept. of Computer Science, Princeton University, Princeton NJ ({felten,li}@cs.princeton.edu)



(a) A prefetching and caching schedule.

(b) A better schedule.

Figure 1: An example of prefetching and caching with advance knowledge of accesses for two disks. One disk holds blocks A, C, E, and F, and another disk holds blocks b and d. Cache size $k = 4$ disks and fetch time $F = 2$ computation steps.

prefetch requires that a buffer be reserved immediately, usually precipitating the replacement of cached data. This earlier replacement may result in an inferior replacement choice, which may actually increase the total number of fetches and degrade performance. Furthermore, in the multiple disk case, poor replacement choices can lead to load imbalance between the disks. The algorithms we study choose different points on the spectrum between aggressive prefetching (with possibly poorer cache replacements) and more conservative prefetching (with closer to optimal cache replacement).

The algorithms explored in this paper also differ in the degree to which they require advance knowledge. *Fixed horizon* exploits global knowledge the least while *reverse aggressive* exploits it the most. Using disk-accurate, trace-driven simulation, this paper's results provide a measure of the potential benefit of using global knowledge.

1.2 Parallel prefetching and caching

Prefetching and caching are even more complicated in a system with multiple disks, not only because it is possible to do multiple prefetches in parallel, but also because appropriate cache replacement strategies can alleviate load imbalance among the disks. 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.

In order to develop intuition for why cache replacement strategies can affect parallel prefetching performance, consider the example shown in figure 1. In this example, the cache holds four blocks and 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. Suppose that it takes two time units to fetch a block from disk, and that the fetches on each disk are serialized. 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.

The application references blocks in the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d, and F. Blocks A, C, E and F are on one disk, and blocks b and d are on the other disk. 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 do not fetch if the evicted block will be referenced before the fetched block. For small caches such as in this figure, the fixed horizon and aggressive algorithms both behave in this way.

Figure 1(a) shows the cache block changes using this approach. The total elapsed time for the sequence is 7 time units. Figure 1(b) shows another prefetching schedule on the same reference pattern 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 time unit.

The example shows that the achievable I/O parallelism of multiple disks can be affected by cache replacement and data placement policies. These are the factors that are addressed by the reverse aggressive algorithm.

1.3 Comparing approaches

The *fixed horizon* algorithm is based on the second Informed Prefetching (TIP2) system of Patterson, Gibson *et al.* [26], which manages allocation of cache space and I/O bandwidth between multiple processes, only some of which are disclosing some or all of their future accesses. TIP2 is designed for the case in which sufficient I/O bandwidth exists to service the request stream without stalling on I/O. The *fixed horizon* algorithm, a restriction of TIP2 to a single hinting process, initiates a fetch for a missing block H references ahead of its reference, where H is the ratio between the average time it takes to read a block from disk and the minimum time it takes to consume a block of data. Patterson*et al.* showed that under the assumption of sufficient bandwidth, this strategy eliminates stalls while placing little stress on system resources. However, *fixed horizon* will not look further than H references into the future for fetches to perform. This can cause it to stall on I/O when there is insufficient disk bandwidth.

In contrast, the *aggressive* and *reverse aggressive* algorithms are designed to take maximum advantage of any amount of I/O parallelism. They use knowledge of future accesses to minimize application elapsed time for both small and large numbers of disks. *Aggressive* prefetches as early as possible, provided that the prefetched block is needed by the application sooner than the block that it will replace. When insufficient bandwidth is available, in particular, it becomes more important to schedule prefetch requests to ensure that no bandwidth is wasted.

Reverse aggressive goes beyond *aggressive*'s use of future knowledge by attempting to balance disk workload through carefully selected replacement decisions. Previously, it was shown theoretically that on any access pattern known in advance, *reverse aggressive*'s elapsed time is close to optimal [16]. It is the only one of these algorithms with this theoretical performance guarantee. However, it is not a practical algorithm. First, it is much more complex than the other algorithms, and second, its decisions depend on information farther in the future than the other algorithms. Nonetheless, its relative performance characteristics are of interest: we would like to understand whether or not the theoretical model we have defined actually gives insight into real system performance. If so, the theoretically near-optimal *reverse aggressive* can be used as a benchmark against which to compare other algorithms. The performance of *reverse aggressive* is our best *a priori* estimate of optimal performance.

The new algorithm *forestall* attempts to combine the best features of the other three algorithms: the good performance of *reverse aggressive* and the simplicity and implementability of *fixed horizon* and *aggressive. Forestall* avoids stalling while still making good (late) replacement decisions by estimating the point at which it needs to begin prefetching in order to prevent stalling.

1.4 Summary of results

In this paper we describe the results of a performance evaluation of the different policies for the d-disk integrated prefetching and caching problem. Our results from trace driven simulation demonstrate the practical performance characteristics of these algorithms. On our traces, we found that:

- All four algorithms significantly outperform demand fetching, even when advance knowledge of the access sequence is used to make optimal replacement decisions in conjunction with demand fetching.

- In compute-bound situations, *fixed horizon* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).

- In I/O-bound situations, *aggressive* and *forestall* have the best performance (which is usually matched by *reverse aggressive*'s).

- In any given situation, one of *fixed horizon* or *aggressive* performs close to the theoretically near-optimal *reverse aggressive.*

- In all situations, *forestall* performs close to *reverse aggressive.*

- When data is well-laid out on the disks (e.g., striped), disk loads are balanced even without careful replacement choices. For this reason, *reverse aggressive* does not significantly outperform the other algorithms.

- *Fixed horizon* consistently places the least I/O load on the disks, due to its conservative fetching and near-optimal replacement choices. *Reverse aggressive* and *forestall* are intermediate between *aggressive* and *fixed horizon.*

- Batching of prefetch requests and disk head scheduling are crucial to the performance of prefetching and caching strategies.

- *Forestall* is a promising new approach that combines the best features of the other three algorithms: good performance regardless of I/O- or compute-boundedness, simplicity, and practicality.

We have focused on a rather narrow range of the input space: the single process, fully-hinted case. Clearly, prefetching and caching algorithms must deal effectively with missing or incorrect hints, as well as multiple simultaneously executing processes. *Fixed horizon*, *aggressive* and *forestall* can all be adapted to deal with these more general situations [5, 26].

1.5 Related work

Caching and prefetching have been known techniques to improve storage hierarchies for many years [2, 12]. In architectures, the work on caching and prefetching has focused on bridging the performance gap between CPU and main memory [29]. Research using caching and prefetching in database systems [9, 23, 10] 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 [12, 20, 14, 22, 30, 4, 13, 7, 26]. The most common prefetching approach is to perform sequential readahead [12, 20, 21]. 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 [11, 30, 23, 10, 13]. Recently, caching and prefetching have also been studied for parallel file systems [11, 18, 25].

Although much work has been done in file caching and prefetching, most of it has considered one or the other in isolation. Recent studies for the single disk case showed [6, 5] 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. For the multidisk case, a theoretical study [16] presented and analyzed *aggressive* and *reverse aggressive.* Other parallel prefetching strategies include one stripe lookahead prefetching on RAID arrays, and Patterson *et al.* [26]'s TIP2 system. The one stripe lookahead benefits only applications that use large files, and would perform little prefetching for other applications. TIP2 uses the *fixed horizon* algorithm we have studied here. Patterson *et al.* [26] also present a cost-benefit technique for controlling buffer allocations for both hinting and non-hinting applications in a multi-process environment.

Previous studies of the algorithms considered here have been incomparable. Differences in hardware, both in the processor and the I/O system, as well as in the benchmarks used to evaluate the algorithms, have made it difficult to understand the differences between them. This paper represents the first direct comparison of these approaches. Using the knowledge learned from this comparison, we have designed a new algorithm that attempts to combine the best features of the previous efforts.

1.6 Organization of the paper

In the next section we describe the first three algorithms and their theoretical basis. In section 3, we describe our simulation framework. In section 4, we present the results of our simulations using the first three algorithms. In section 5 we describe the new algorithm *forestall* and present simulation results on its performance. We present our conclusions in section 6.

2 The algorithms

We begin by introducing the framework used to study this problem and the terminology used in the rest of the paper.

2.1 Theoretical model

Our theoretical model consists of two levels of memory hierarchy: a cache of K data blocks, and d (disk) storage devices. The execution of a program makes a known *request sequence* of references $r_1, r_2, \ldots r_n$ to a set of m data blocks.

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.

We assume that each block resides on a single disk. Fetches to a single disk are serialized, but fetches on different disks can be executed concurrently.

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; this is equal to the number of references plus the total stall time.

The goal is to minimize application elapsed time, by deciding 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).

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 a 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, that all disk accesses take the same amount of time, and that there is no CPU overhead incurred by issuing an I/O request. These simplifications were made in order to make the problem theoretically tractable. Our simulations use actual CPU times collected in our traces and an accurate simulation model of modern disk drives, and charge a driver overhead for each request made to a disk.

2.2 Optimal prefetching rules

The following simple rules can be assumed of any optimal strategy in the single-disk case [6]. [1]

- *Optimal fetching*: when fetching, always fetch the missing block that will be referenced soonest;
- *Optimal replacement*: when fetching, always evict the block in the cache whose next reference is furthest in the future;
- *Do no harm*: never evict block A to fetch block B when A's next reference is before B's next reference;
- *First opportunity*: never evict A to fetch B when the same thing could have been done one time unit earlier.

Unfortunately, as exhibited in the example in section 1, some of these rules no longer hold in the multiple-disk case. It may be necessary to violate all of the rules except *first opportunity* to produce an optimal schedule.

2.3 The *fixed horizon* algorithm

As described earlier, the *fixed horizon* algorithm is based on the TIP2 system running a single hinting process [26].

Fixed horizon: Whenever there is a missing block at most H references in the future, issue a fetch for that block, replacing the cached block whose next reference is furthest in the future, provided that reference is further than H accesses in the future (which will certainly hold if $H < K$).

Fixed horizon is consistent with the first three rules of optimal prefetching for a single disk. An advantage of not following the fourth rule is that *fixed horizon* needs less information about references beyond the prefetch horizon than the other algorithms. A disadvantage is that when additional

[1] These rules are optimal in the sense that any schedule that does not follow them can be transformed into one that does, with performance at least as good.

information is available, *fixed horizon* can have elapsed time nearly twice optimal.

The prefetch horizon H is computed as the ratio of the average time it takes to read a block from disk and the minimum time it takes to consume a single block of data. In the theoretical model, $H = F$.

Fixed horizon tries to fetch as late as possible without stalling in order to make the best possible replacement decision. Each fetch is issued so that it will complete just in time for the reference. If parallelism increases to the point that each request is made to an idle disk, this algorithm is optimal. However, in practice, a sufficient number of disks may not be available. In this case, *fixed horizon* may initiate fetches too late to avoid stalling. In fact, because it never initiates a fetch more than H references ahead of the missing block, *fixed horizon* may allow a disk to become idle even though the future requests beyond the prefetch horizon contain many missing blocks. On the other hand, if the missing blocks in the sequence tend to be separated by many intervening references to blocks that are present in the cache, we'd expect *fixed horizon* to have performance much closer to optimal than its worst case.

2.4 The *aggressive* algorithm

The (multi-disk) aggressive algorithm is based on the Cao *et al.* (single-disk) aggressive algorithm [6], which is provably near-optimal in the single-disk case.

(Multi-disk) aggressive: Whenever a disk is free, prefetch the first missing block on that disk, replacing the block whose next reference is furthest in the future, under the condition that the next access to the evicted block is after the next access to the block being fetched.

Aggressive is the most aggressive prefetching strategy that is consistent with the four optimal prefetching rules described in section 2.2. As mentioned, some of these rules are no longer valid in the multiple disk case. This provides some of the intuition for the following theorem.

Theorem 1 *[16] For any access pattern, and any layout of data on disks, the elapsed time of* aggressive *is at most* $d(1+\epsilon_1)$ *times that of the optimal elapsed time (the minimum possible), where* d *is the number of disks, and* ϵ_1 *is a small constant that depends on system parameters.*[2]

There are worst case access patterns/data layouts for which the elapsed time of aggressive *is at least* d *times the minimum possible.*

It is important to note that this worst case result depends on access patterns and data layouts in which the load is heavily unbalanced between the disks. If the request sequence is balanced, *aggressive* has near-optimal performance.

2.5 The *reverse aggressive* algorithm

The *reverse aggressive* algorithm exploits global knowledge in order to produce a prefetching schedule that achieves near-optimal elapsed time. It does this by balancing disk workload through carefully selected replacement decisions.

Reverse aggressive: Construct a prefetching schedule for the *reversed* sequence that *replaces* at most one block on each disk in parallel as follows: Whenever a disk is free, determine the block B not needed for the longest time on that disk. If the next request to B is after the first missing block, issue a fetch for the missing block, *replacing* B. Transform this prefetching schedule back to a schedule for the original sequence by treating each fetch on the reverse sequence as an eviction on the forward sequence and vice versa.

For a proof of correctness, more details on how and why this algorithm works well, and a proof of the following theorem, see [16].

Theorem 2 *[16] For any request sequence, and for any layout of the data on the disks, the elapsed time of reverse aggressive is at most* $1 + \epsilon_2$ *times the optimal elapsed time.*[3]

There are two key properties of *reverse aggressive* that result in this theorem. First, whereas *aggressive* chooses evictions without considering the relative loads on the disks, *reverse aggressive* greedily evicts to as many disks as possible on the reverse sequence. In the forward direction, this translates to performing a maximal set of fetches in parallel. The fact that these are fetches in the forward direction means that at some point earlier in the sequence, corresponding blocks were evicted. Thus the eviction decisions of *reverse aggressive* on the forward sequence are based on the ability to prefetch the evicted blocks later on in parallel. Second, whereas *aggressive* can wastefully prefetch ahead on some of its disks, *reverse aggressive* is greedy in the reverse direction. Consequently, it is fetching blocks in the forward direction just in time (to the extent possible) for them to be used. This results in performing close to the best evictions possible for those fetches.

2.6 Practical considerations

Several important features of real systems are not captured by our theoretical model.

1. Disk response times and CPU times between I/O requests are not constant.

   We use average values for each and expect that variation in event times does not substantially invalidate the algorithm's decisions. In our experimentation, this does not appear to be a major effect, with one exception (see section 4.3). (The systematic effects of disk scheduling on disk response time are considered separately).

[2] ϵ_1 here is F/K where F is the fetch time/compute time ratio and K is the cache size measured in blocks. For typical system parameters ϵ_1 is less than 0.02.

[3] ϵ_2 here is less than Fd/K, where F is the fetch time/compute time ratio, d is the number of disks, and K cache size in blocks. For typical sytem parameters, ϵ_2 is less than 0.1, and sometimes significantly less.

2. Access patterns exhibit locality of reference, and loads are balanced across the multiple disks when data is striped.

   In practice, this allows both *fixed horizon* and *aggressive* to effectively utilize multiple disks, and achieve elapsed times comparable to the theoretically superior *reverse aggressive*.

3. Disk accesses require significant CPU overhead to form the request, communicate with the disk, and service the resulting interrupt(s). Thus, avoidable data fetches may add elapsed time even if they do not cause stalls.

   Because the theory assumes that fetches entail no CPU overhead, this penalty punishes overly aggressive fetching. In practice, this effect favors the *fixed horizon* algorithm since its late replacement decisions tend to lead to the fewest fetches.

4. Disk response time is sensitive to the order in which requests are serviced.

   In particular, disk scheduling reduces average disk response time as more accesses are presented and allowed to be reordered by the disk (driver). Although *fixed horizon* implicitly allows multiple outstanding requests at each disk, *aggressive* and *reverse aggressive* were defined to submit only one request at a time, since in the theoretical model there is no advantage to batching. Because of the significance of the disk scheduling effect, we modify the definitions of *aggressive* and *reverse aggressive* to submit disk requests in batches. We have found that the performance of all three algorithms benefits from the CSCAN disk scheduling algorithm.

Reverse aggressive also benefits from batching of requests during its construction of its prefetching schedule (the reverse pass over the request sequence). This is because typical request sequences exhibit spatial locality; by batching requests on the reverse pass, *reverse aggressive* generates missing blocks to be fetched on the forward sequence in groups that exhibit locality of reference.

The inter-request CPU time is actually composed of two components, a fixed amount of time to read a block out of the cache, and a variable amount of time to process the data. Our implementation of *fixed horizon* assumes the data processing time to be zero, and uses the ratio of the average disk response time to the time to read a block from the cache as the prefetch horizon H. This ensures that any prefetch to an idle disk will complete in time for the reference. Assuming an average disk response time of $15ms$ (which is usually an overestimate in our simulations) and $243\mu s$ to read a block from the cache (which was measured on the implemented TIP2 system) yields a value of $H = 62$; we used this value in all our simulations, except where noted otherwise.

2.7 Implementations of the algorithms

In the context of the considerations of the previous section, we summarize the implementations we compared.

Fixed horizon: Whenever there is a missing block at most H references away, issue a fetch for that block, replacing the block whose next reference is furthest in the future. Note that this algorithm may at any time have up to H outstanding references to a disk yielding opportunities for disk scheduling.

Aggressive: Whenever a disk D is free, construct a batch of at most `batch-size`[4] fetches to initiate on D as follows: As long as the first missing block B on disk D precedes the block B' whose next request is furthest in the future, add the fetch/eviction pair B/B' to the batch. Issue the batch.

If two or more disks are free at the same time, we consider all their missing blocks together, in order of increasing request index. Each next missing block is issued to the appropriate disk (and the best possible choice of evictions is made), if the disk's batch is not full and the *do no harm* rule allows it. At some point, either the last free disk's batch becomes full or the *do no harm* rule disallows issuing further requests.

Reverse aggressive: Assuming a fixed ratio F between the time for a disk access and the inter-reference CPU time, consider the reversed sequence, and use it to derive a prefetching schedule as described in section 2.5, but construct the schedule in batches as done by *aggressive*.

This prefetching schedule is then transformed into a schedule of fetch/eviction pairs for the forward sequence. Associated with each eviction is a *release time*, the earliest index in the request sequence at which the block can be evicted (i.e. one greater than the index of the last request to the block until it is possibly fetched back into the cache at some later time.) The eviction choices are naturally ordered by increasing release point due to the method used by *reverse aggressive* to construct its schedule. Fetches may need to be re-ordered according to increasing request index; they are then matched to eviction choices according to these orderings.

This schedule is used to drive the disk model as follows. Whenever a disk D is free, add the first up to `batch-size` fetch/eviction pairs B_i/B'_i that have been released, and for which B_i resides on disk D, to the batch. Issue the batch.[5]

Notice that *aggressive* and *fixed horizon* use less lookahead information than *reverse aggressive*, in that for both of them, the "only" future information needed are the identities of the next missing blocks (up to H missing blocks for *fixed horizon*, and up to d times `batch-size` for *aggressive*), and their positions in the sequence relative to the next references to blocks currently in the cache.

3 Simulation framework

We used trace-driven simulation to evaluate the performance of the algorithms. We believe our simulation model to be an accurate reflection of the practical performance characteristics of the algorithms. The reference streams are taken from traces of real applications' behavior; the trace information

[4] The batch sizes used are listed in table 6.

[5] The batch sizes and estimate F used by *reverse aggressive* are discussed in section 4.4.

Sector size	sectors per track	tracks per cylinder
512 bytes	72	19
cylinders	rotational speed	disk cache size
1962	4002 rpm	128 Kbytes
ave. access time (8Kbyte)	controller interface	transfer rate
22.8ms	SCSI-II	10 MB/sec

Table 1: HP 97560 characteristics.

xds elapsed times (secs)				
	CMU simulator		UW simulator	
disks	F.H.	Agg.	F.H.	Agg.
1	63.3	61.6	65.6	63.7
2	36.9	34.1	38.0	34.3
3	33.6	33.9	36.2	33.7
4	33.8	35.1	34.2	35.1
5	33.0	34.2	33.5	34.4

synth elapsed times (secs)				
	CMU simulator		UW simulator	
disks	F.H.	Agg.	F.H.	Agg.
1	213.0	168.5	201.4	155.8
2	136.3	126.9	130.9	121.7
3	118.9	149.5	118.9	150.4
4	118.9	150.4	118.9	150.1

Table 2: Comparison of the simulators on the xds and synth traces.

we use is unaffected by prefetching and caching activity. The accurate modelling of disk fetch times, I/O driver overhead costs, and application process compute times in the simulations is a key difference relative to the theoretical framework. However, our simulators do not model serialization of DMA transactions. [6]

Two separate simulators were developed, one at Washington (UW) and one at Carnegie Mellon (CMU). The UW simulator uses the disk drive simulation of Kotz *et al.* [19] (which is based on that of Ruemmler and Wilkes [27]) to accurately model I/O costs. This simulation models fine architectural details to provide a very accurate simulation of the HP 97560 disk drive. Table 1 lists several characteristics of the HP 97560 (taken from [27]). The CMU simulator uses the Berkeley RaidSim [8] simulator, as modified at CMU, to simulate 0661 IBM Lightning disk drives.

The simulators were cross-validated on a common set of traces. The CMU simulator does not implement *reverse aggressive.* We obtained good agreement between the simulators on the results for *aggressive* and *fixed horizon* for several traces. Table 2 shows the elapsed times measured by the simulators for the xds and synth traces described below. Remaining differences between the simulators are consistent with the differences in the disk models. We report here results for all algorithms obtained using the UW simulator.

In our simulations, we ignore write operations. Write performance is less critical to I/O performance since the application generally does not have to wait for the disk to be written. Moreover, the impact this has on the results is small since most of the references in our traces are reads.

trace	reads	distinct blocks	compute time (sec)
dinero	8867	986	103.5
cscope1	8673	1073	24.9
cscope2	20206	2462	37.1
cscope3	30200	3910	74.1
glimpse	27981	5247	38.7
ld	5881	2882	8.2
postgres-join	8896	3793	11.5
postgres-select	5044	3085	79.2
xds	10435	5392	30.8
synth	100000	2000	99.9

Table 3: Trace summary data.

We simulated disk arrays of sizes 1-8, 10, 12, and 16. Most of our figures show a smaller range of sizes, however. In each case, the performance with a larger number of disks is the same as that with the largest number of disks shown.

3.1 File access traces

We used a set of traces collected on a DECstation 5000/200. The running time of all the applications is dominated by disk read accesses. Each trace consists of a sequence of file block read requests in the order they were issued, and the sequence of measured process compute times between read requests, of a single execution thread. We used an I/O driver overhead of .5ms per I/O operation, which is typical of the 5000/200.

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. With multiple queries, cscope will read multiple files sequentially multiple times.

dinero: a cache simulator written by Mark Hill. This application reads one file sequentially multiple times.

glimpse: a text information retrieval system from the University of Arizona, searching for four keywords in a 40MB snapshot of news articles. It builds approximate indexes for words to allow both relatively fast search and small index files. The result is that the index files are accessed repeatedly, whereas the data files are accessed infrequently.

postgres-join: the Postgres relational database system 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 [3]. Since the result relation is small, most of the file accesses are reads. Here, the index blocks are accessed much more frequently than the data blocks.

[6] We do not expect this to have a significant effect on the results since the DMA time is much less than the disk access time.

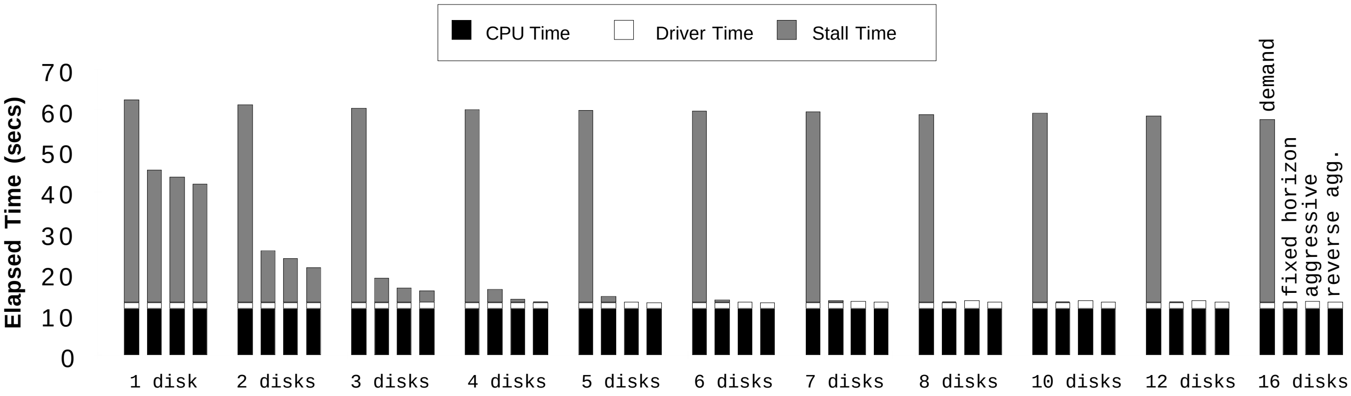


Figure 2: Performance on the postgres-select trace. Each group of bars represents the performance of the four algorithms *optimal demand fetching*, *fixed horizon*, *aggressive*, and *reverse aggressive*, in left-to-right order.

postgres-select: 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 [3] and uses indexed search.

ld: the Ultrix link-editor, building the Ultrix 4.3 kernel from about 25MB of object files.

xds: a 3-D data visualization program, XDataSlice, generating 25 planar slice images at random orientations from a 64MB data file.

Finally, we used a synthetic trace **synth** containing 50 passes through a loop of 2000 sequential blocks. Compute times between read requests were generated according to a Poisson distribution with a 1 ms mean.

Table 3 shows the length (number of read requests), number of distinct blocks requested, and total application compute times for each of the traces.

The cache size was set to be 10MB (or $K = 1280$ blocks of 8 kbytes each) for all traces except dinero and cscope1. These traces contain references to fewer than 1280 distinct blocks. For these traces, the cache size was reduced to 4MB (512 blocks). We assume the cache to be empty (or to contain some other application's data) when the traced application starts. The entire cache is available to the traced application.

3.2 Data placement and disk head scheduling

The data was striped across the array using a one-block stripe unit. Some of our traces represented block numbers by (file,offset) pairs; for these we chose a random starting point within a group of 8550 8kbyte blocks (which occupy 100 cylinders on the HP 97560) for each file, corresponding to typical file system clustering mechanisms. The maximum seek time within a group of 100 cylinders is 7.24ms. Thus, in our simulations the average response time is typically lower than the 22.8ms listed in table 1. Other traces referred to logical filesystem block numbers; for these traces we used the actual block number for each access. Except where noted, we use CSCAN disk head scheduling.

4 Results

In the following sections, we examine the behaviors of the algorithms in detail. We begin by comparing the performance of the algorithms with that of demand fetching. We then examine the algorithms' performance on the synthetic trace, an easily understood access pattern that illustrates the key differences in behavior between the algorithms. Next we examine performance on the application traces, and explore the effects on the results of changes in various simulation parameters.

4.1 Comparison with demand fetching

In order to make this comparison as favorable as possible to demand fetching, we use the optimal offline replacement policy: whenever a block is fetched, the block in the cache whose next reference is furthest in the future is replaced. Figure 2 shows the elapsed times of the three algorithms and of optimal demand fetching on the postgres-select trace for varying numbers of disks between one and sixteen. The elapsed times are divided into three components: process compute time, I/O driver overhead (processor) time, and the time the processor spends idle, stalling on I/O. From this figure we see that (1) all three prefetching algorithms significantly outperform optimal demand fetching, and (2) the three prefetching algorithms achieve near linear reduction in I/O overhead until the applications become compute-bound. These two behaviors are consistent across all the applications we have studied.

4.2 Fundamental differences

The synthetic trace is used to examine the algorithms' behavior on a simple, known sequence in order to gain insight into the algorithms' performance. This trace shows the relative behaviors typical of the three algorithms in exaggerated form. Figure 3 summarizes the results for one to four disks.

The sequential accesses allow excellent performance from the disks; average response times are between 3 and 4 ms. In each case, *fixed horizon* performs 38000 fetches, 720 more

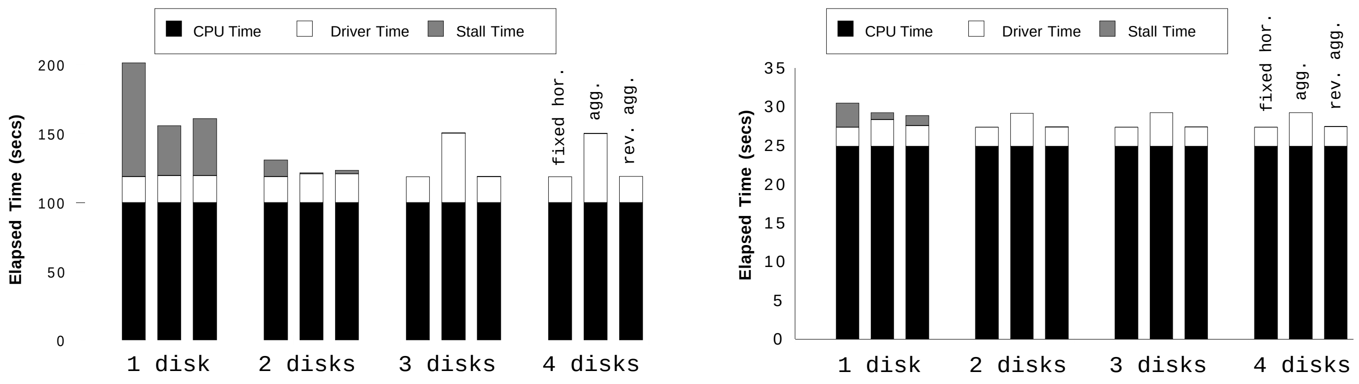


Figure 3: Performance on the synth (left) and cscope1 (right) traces. Each group of bars represents the performance of the three algorithms *fixed horizon*, *aggressive*, and *reverse aggressive*, in left-to-right order.

than the minimum possible 37280 performed by optimal demand fetching. (The total sequence length is 100,000).

With a single disk, the synthetic application is I/O bound. *Fixed horizon*'s conservative prefetching strategy reduces I/O stalling relative to demand fetching, but not as much as *aggressive*'s and *reverse aggressive*'s more aggressive strategies. After each pass through the loop under *fixed horizon*, the cache contains 1280 sequential blocks and the other 720 blocks in the sequence are not cached. The clustering of the 720 missing blocks allows good disk performance; however, the clustering of the 1280 cached blocks causes *fixed horizon* to leave the disk idle until the last H cached blocks are being read. *Aggressive* and *reverse aggressive* perform 39240 and 39265 fetches, respectively, slightly more fetches than *fixed horizon*'s 38000, resulting in a small difference in driver overhead. However, they are able to eliminate much of the I/O stall time by prefetching distant blocks and thus not idling the disk appreciably.

With two disks, *fixed horizon* is able to eliminate most of the stall time, without increasing the total number of fetches. *Aggressive* has nearly eliminated stall time completely, but at a higher driver cost due to its increased number (41902) of fetches. *Reverse aggressive* is between *fixed horizon* and *aggressive* in stall time; it performs 42000 fetches. Elapsed times are similar under all three algorithms. This case marks the transition from I/O-boundedness to compute-boundedness.

With three disks, stall time has been eliminated completely by all three algorithms. *Aggressive* uses the excess I/O bandwidth to prefetch and subsequently evict every block for every reference. In fact, because *aggressive* is willing to prefetch significantly ahead on one disk relative to others, it wastes 994 fetches, replacing a prefetched block from the cache before it is used in order to fetch a block on a different disk that will be needed sooner. Fortunately, this effect does not increase as the number of disks increases since with increasing I/O bandwidth, *aggressive*'s prefetching becomes so successful that every fetch is to the first missing block in the future. Such a block can never be replaced before it is used, since that would violate the *do-no-harm* rule.

Nonetheless, the elimination of stall time by *aggressive* comes at a high cost: the driver overhead for the extra fetches pushes *aggressive*'s elapsed time higher than the two-disk case. In contrast, *fixed horizon* prefetches far enough ahead to serve all requests without stall, but no farther. Dedicating at most H buffers to prefetching, *fixed horizon* is able to eliminate stalling altogether without any additional fetches. *Reverse aggressive* performs 37907 fetches, fewer than *fixed horizon*, also eliminating stall time.

4.3 Application traces

The application traces show differences among the three algorithms similar to those shown by the synthetic trace, but less pronounced.

The right portion of figure 3 shows the performance of the three algorithms on the CPU-bound cscope1 trace. The behavior here is similar to that for the synthetic trace: *aggressive* eliminates stalling but issues too many fetches resulting in a greater driver overhead.

At the I/O-bound end of the spectrum, figure 4 shows a detailed breakdown of the performance of the three algorithms on the ld trace, from one to sixteen disks. With one disk, all three algorithms are I/O bound and have comparable performance. From two to eight disks, the more aggressive prefetching of *aggressive* and *reverse aggressive* results in somewhat less stalling than *fixed horizon.* At ten disks, *fixed horizon*'s performance matches *aggressive*'s. Beyond this point, the tradeoff between excessive stalling caused by leaving disks idle, and excessive driver overhead caused by prefetching aggressively, favors *fixed horizon* over *aggressive.* The other traces reflect similar trends, with different points of crossover: above five disks for postgres-select, glimpse, and cscope2, and below five disks for postgres-join, dinero, cscope1, and xds.

An exception to the generally best performance of *reverse aggressive* is the cscope3 trace, shown in figure 5. Note that *reverse aggressive*'s performance is much worse than *aggressive*'s with one disk. This is a case in which the differences between the theoretical model and the simulation model affect the performance of *reverse aggressive.* Recall that since *reverse aggressive* is offline, it generates a complete schedule based on its *estimate* of F. When it uses a smaller estimate of F, each fetch is assumed to complete earlier (relative to

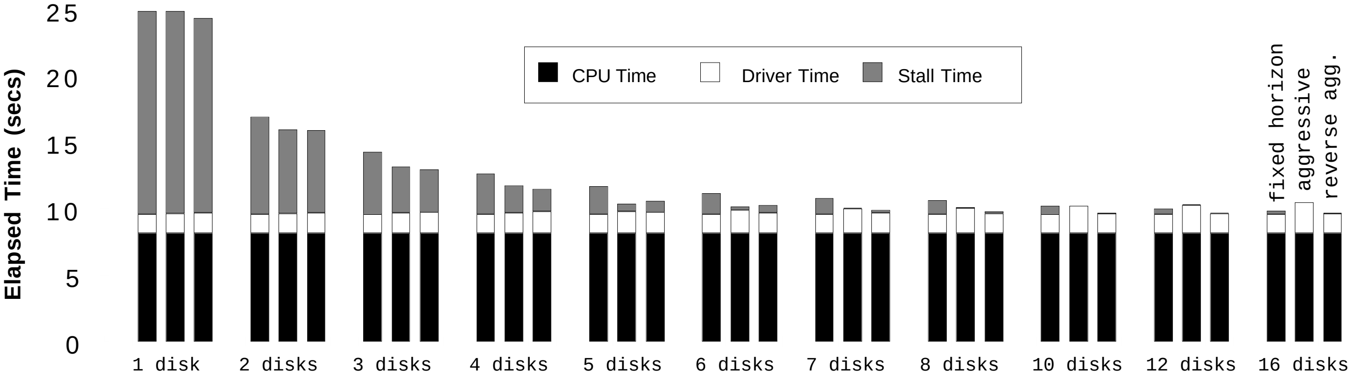


Figure 4: Performance on the ld trace.

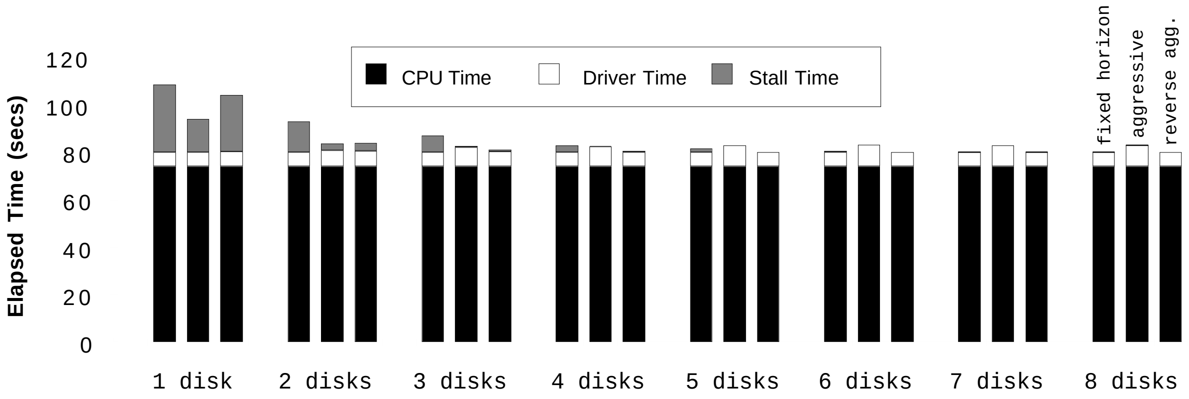


Figure 5: Performance on the cscope3 trace.

the inter-reference compute time) and therefore *reverse aggressive* generates a more aggressive prefetching schedule that keeps the disk(s) busier. When it uses a larger estimate of F, each fetch is assumed to take longer, and therefore *reverse aggressive* must delay the scheduling of subsequent fetches in the sequence, thus generating a more conservative prefetching schedule. In our implementation of *reverse aggressive*, the single best estimate of F is used for each trace. On traces with large variation in inter-reference compute times, any single estimate of F will be either too small or too large for some parts of the trace. This is the case for cscope3 – examination of the trace reveals that the inter-reference compute times are bursty. Runs of compute times near 1ms are interspersed with runs of times around 7ms. Since the average fetch time on this trace with one disk is about 8ms, the ratio of fetch time to compute time (the "true" value of F) varies from about 1 to about 8.

In fact, with a single disk, *aggressive* has the same theoretical performance bounds as *reverse aggressive*. It is not surprising that *aggressive*'s inherent adaptivity to varying fetch times and compute times should give it an advantage over *reverse aggressive* in this case. This effect is noticable, but less pronounced, on the synth trace as well.

On the remaining traces, *reverse aggressive*'s elapsed time varies from 3.6% worse to 10.7% better than the superior of *fixed horizon* and *aggressive* in any given configuration. For the full data, see [17].

disks	demand fetching	*fixed horizon*	*aggressive*	*reverse aggressive*
1	.82	.98	.99	.98
2	.41	.90	.92	.92
3	.27	.82	.87	.85
4	.20	.72	.81	.80
5	.16	.66	.70	.69
6	.13	.58	.63	.60
7	.12	.50	.62	.50
8	.10	.45	.56	.42
10	.08	.36	.43	.35
12	.07	.30	.35	.28
16	.05	.22	.26	.21

Table 4: Disk utilization on the postgres-select trace.

Table 4 shows the utilization of the disks (averaged over the disks when there are more than one) for demand fetching and the three prefetching algorithms on the postgres-select trace. For moderate numbers of disks, *aggressive* places the greatest load on the disks, followed by *reverse aggressive* and then *fixed horizon*; demand fetching places the least load on the disks. With a very high degree of disk parallelism, *reverse aggressive*'s offline schedule places even less load on the disks than *fixed horizon*'s conservative strategy.

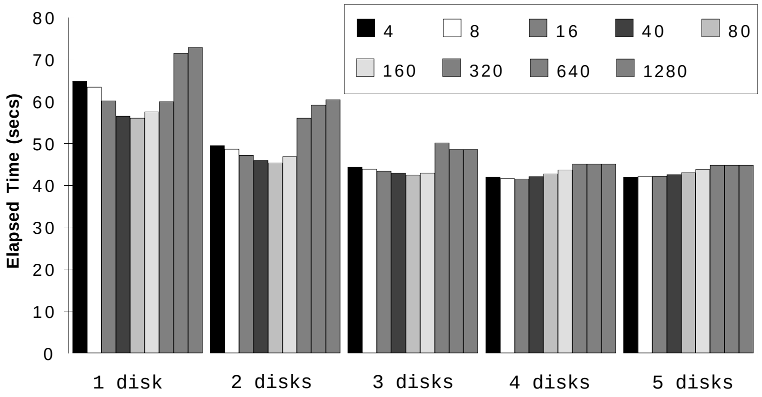


Figure 6: Performance of *aggressive* on the cscope2 trace, as a function of the batch size.

4.4 Varying parameters

The performance of the algorithms depends on a set of parameters which interact in complicated ways with the applications' access patterns and inter-reference compute times, the layout of data on disks, the disk-scheduling discipline, and the characteristics of the disks. In this section, we explore the behavior of the algorithms when some of these parameters are varied. For lack of space, we present general observations and only a small portion of the data. For the full data, see [17].

We have already described most of the primary effects that explain what we see. These are:

- *scheduling*: an increase in the number of outstanding fetches issued by a prefetching algorithm results in increased latitude to reorder fetches and thus reduced disk response times. This effect is strongest in I/O-bound situations.
- *out-of-order fetching*: reordering of fetches can increase stall penalties when early missing blocks are fetched after later missing blocks. This effect is strongest in CPU-bound situations where any stall penalty is costly. When there is significant stalling, this effect is masked by other stalls and compensated for by the reduced average response time.
- *early replacement*: as prefetching becomes more aggressive, inferior replacement choices are made, leading to more fetches and in many cases, an increase in elapsed time.
- *limited aggressiveness*: the extent to which an algorithm can prefetch is limited by the *do no harm* rule.

Disk-head scheduling

The results shown in the previous section were obtained using CSCAN disk-head scheduling. CSCAN was used rather than SCAN since the HP 97560 contains a readahead buffer; CSCAN always scans in the same direction that the disk reads, improving the hit rate in the readahead buffer. We compared the performance impact of CSCAN disk-head scheduling versus FCFS scheduling. Relative to FCFS, CSCAN improves the performance of *reverse aggressive* the most, up to 24%, and that of *fixed horizon* the least, up to 15%. For *aggressive*, the greatest benefit was 19%. Because of out-of-order fetching, CSCAN sometimes degrades performance slightly relative to FCFS in compute-bound situations. This effect is strongest for *fixed horizon* since it issues fetches later than they are issued by the other algorithms. The maximum degradation we observed is 3.6% (for *fixed horizon* with six disks on the glimpse trace).

disks	*fixed horizon*	*aggressive*	*reverse aggressive*
1	14.9	19.2	24.0
2	4.85	11.3	22.1
3	2.59	8.36	19.9
4	0.53	3.59	6.71
5	-0.62	-0.77	0.0
6	-0.68	-0.31	0.0
7	-2.15	-0.45	0.0
8	-0.42	-0.17	0.0
10	-0.05	0.09	0.0
12	0.0	0.11	0.0
16	0.0	0.0	0.0

Table 5: Percentage improvement of CSCAN over FCFS on the postgres-select trace.

Table 5 shows the performance benefit of CSCAN scheduling relative to FCFS on the postgres-select trace for all three algorithms with 1-16 disks.

The batch size used by *aggressive*

Figure 6 shows the effect of varying *aggressive*'s batch size on the cscope2 trace. For each number of disks, performance initially improves with increasing batch size due to improved scheduling. For example, for one disk, the average fetch time drops from 10.4ms to 8.4ms as the batch size increases from 4 to 160. Eventually, out-of-order fetching and early replacement become more important and performance drops off again. For example, for one disk the number of fetches increases from 6771 to 9806 as the batch size increases from 160 to 1280.

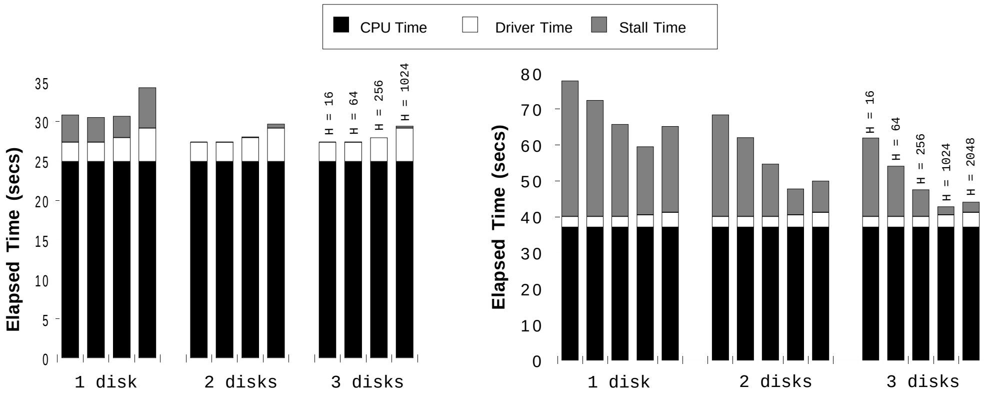


Figure 7: Performance of *fixed horizon* as a function of the prefetch horizon H on the cscope1 (left) and cscope2 (right) traces.

1 disk	2 disks	3 disks	4 disks
80	40	40	16
5 disks	6 disks	7 disks	> 7 disks
16	8	8	4

Table 6: Batch sizes used for *aggressive*.

As the number of disks increases, the variation in performance with batch size diminishes, and the best batch size shifts to a smaller value. This is because in more compute-bound situations, out-of-order fetching and limited aggressiveness are the dominant factors. Because of limited aggressiveness, the number of fetches increases only from 11325 to 11399 as batch size increases from 160 to 1280 with 5 disks.

Although the optimal batch size decreases with the number of disks for all the traces, it varies significantly from trace to trace. For example, for the xds trace, the optimal batch size for one to three disks was 16, and for four or more was 4. All the results for *aggressive* presented in section 4.3 were obtained using the batch sizes given in table 6. The performance of *aggressive* with these fixed batch sizes is on average 0.7 % worse (and at most 11% worse) than its performance with the best batch size for the configuration.

Prefetch horizon

The left side of figure 7 shows the effect of varying *fixed horizon*'s prefetch horizon H on the cscope1 trace. We see that for each number of disks, performance deteriorates with increasing H (except on one disk, where it improves slightly until $H = 64$ is reached). This is due to out-of-order fetching and early replacement. For example, with 1 disk, earlier replacements cause the number of fetches to increase from 4959 with $H = 64$ to 8535 with $H = 2048$. Out-of-order fetching accounts for all the stall time with 2 and 3 disks when $H \geq 512$; using FCFS scheduling this stall time is eliminated.

On the more I/O bound traces such as cscope2, also shown in figure 7, we find a significant initial performance improvement with increasing H because the more aggressive prefetching eliminates stalling. Only at very large values of H does performance decline again.

The parameters used by *reverse aggressive*

We experimented with the batch size and fixed value of F used by *reverse aggressive* to construct its schedule on its reverse pass over the request sequence, as well as the batch size used on the forward pass. Since we use *reverse aggressive* only as a benchmark against which to compare the other algorithms, the main purpose of these experiments was to determine the optimal configuration (choice of F and batch sizes) for each trace, for each number of disks.

These experiments show that, as with *aggressive*, a smaller (resp. larger) batch size benefits a more compute-bound (resp. I/O-bound) application. Recalling that as *reverse aggressive*'s estimate of F decreases, it becomes increasingly aggressive, we similarly find that a smaller (resp. larger) value of F benefits a more I/O-bound (resp. compute-bound) application.

Processor speed and cache size

In order to assess the impact of improved CPU performance relative to disk performance, we ran our trace-driven simulations assuming a processor twice as fast. For these tests, *fixed horizon*'s prefetch horizon H was doubled to 124. The results are entirely unsurprising: faster processors are more dependent on I/O performance so that the payoff of using multiple disks and prefetching is increased. In addition, since a larger number of disks is needed to eliminate I/O overhead, the point at which the tradeoffs begin to favor *fixed horizon* over *aggressive* is shifted to a larger number of disks. This behavior was consistent across the applications.

In order to assess the impact of cache size on performance, we ran our trace-driven simulations with varying cache sizes: 640, 1280, and 1920 blocks. As cache size increases, the performance of all the algorithms improves. In I/O-bound cases, a larger cache improves *aggressive*'s and *reverse aggressive*'s performance more than *fixed horizon*'s since they prefetch more aggressively. In more compute-bound cases, *aggressive*'s excessive driver overhead penalizes it even more with a larger cache, so that *fixed horizon*'s performance relative to *aggressive* improves slightly as cache size increases. This is illustrated in table 7, which shows the performance of *fixed horizon* relative to *aggressive* as percentage difference, as a

cache size	1 disk	2 disks	4 disks	8 disks	16 disks
640	6.0	14.7	24.8	7.3	-2.6
1280	11.3	20.2	24.5	5.7	-3.8
1920	13.8	25.0	21.7	5.7	-3.8

Table 7: Elapsed time as a function of the cache size and number of disks of *fixed horizon* relative to *aggressive* (percentage difference) on the glimpse trace.

function of the cache size and the number of disks on the glimpse trace.

5 A new approach

We have designed a new algorithm, *forestall*, attempting to combine the best features of all three previously described algorithms: the good performance of *reverse aggressive* regardless of I/O-boundedness or compute-boundedness, and the simplicity and implementability of *fixed horizon* and *aggressive*. *Forestall* tries to avoid stalling while still making good (late) replacement decisions by estimating the point at which it needs to begin prefetching in order to prevent stalling. It makes this estimate based on its current cache state.

Returning to the theoretical model, suppose that there is a single disk, and that at some point during the servicing of the request sequence, the cache contains the next $2F-1$ blocks requested. (Recall that in the theoretical model, the interreference CPU time is taken to be 1 time unit, and the time to fetch a block from disk is F time units.) Further suppose that the subsequent two requests are missing from the cache. *Aggressive* immediately starts fetching and avoids stalling on the missing blocks, bringing the second missing block into the cache at time $2F$ – just in time to serve the request without stalling. *Fixed horizon* leaves its disk idle until the cursor is within F requests of the first missing block; it stalls $F-1$ steps on the second missing block. In contrast, suppose there is only one missing block at a distance of $2F-1$ from the cursor. In this case, *aggressive* will fetch immediately and make a possibly inferior replacement choice. *Fixed horizon* waits until its cursor is within F steps of the missing block, and prefetches just early enough to avoid stalling; in the intervening time, it may have finished using a block that is not needed until later in the sequence (if at all) than the one evicted from the cache by *aggressive*.

Forestall behaves as does *aggressive* in the first case, and as does *fixed horizon* in the second. For each i, $i \geq 1$, let d_i denote the distance from the cursor to the i^{th} missing block in the request sequence. For any $i \geq 1$, if $iF > d_i$, processing will surely stall on the i^{th} missing block or some earlier missing block. It will take iF time units to fetch the first i missing blocks, and at most the next d_i requests can be served concurrently. *Forestall* initiates a prefetch according to the *optimal fetching* and *optimal replacement* rules whenever $iF \geq d_i$ is true for some i and the *do no harm* rule allows it.

Practical considerations

As do the other algorithms, *forestall* requires modifications in order to account for differences between the theoretical model and real systems. Requests need to be issued in batches in order to reduce average disk access times. The ratio F of disk response time to interaccess time is not constant and must be estimated. In our implementation, we estimate F by tracking recent disk response times and compute times: F is dynamically computed on a per-disk basis as the ratio between the sum of the most recent 100 disk access times and the most recent 100 interreference CPU times.

Just as we needed the prefetch horizon H to be an overestimate of F for *fixed horizon* to have adequate performance, *forestall*'s performance depends on overestimating F in certain situations as well. We denote by F' the overestimate of F used by *forestall*. We evaluated *forestall*'s performance with different values of the parameter F'. We found that the best choice of F' depended on the per-trace average disk access times. For those traces for which the average disk access time was small, in the 3-4ms range, it was best to take $F' = F$. For those traces for which the average disk access time was larger, it was best to take $F' = 4F$. This is not hard to explain. Traces with disk access times in the 3-4ms range must contain a great deal of sequential access, so that most requests hit in the disk's readahead cache and are served by the CSCAN scheduler in the order in which they are received. When this happens, it is not necessary to prefetch aggressively. When the disk access times are large, the access pattern is more complicated, and disk access times more varied. *Forestall*'s mechanism for deciding when to prefetch benefits from overestimating the potential to stall. This smooths out the variations and avoids stalling due to the reordering of requests by CSCAN. Our implementation of *forestall* adapts to the observed disk access times, using the small value of F' for small disk access times (less than 5ms on average), and the larger value of F' for larger disk access times. Finally, because of the reordering of requests by CSCAN, we found it necessary to add *fixed horizon*'s rule to issue a fetch whenever the cursor is within H requests of a missing block. This avoids stalling on reordered requests in situations in which the $iF' \geq d_i$ rule delays fetching until the cursor is very near the first missing block.

Rather than using complete lookahead information in our implementation of *forestall*, we check the value of the expression $iF - d_i$ only for those missing blocks within distance $2K$ of the cursor, where K is the cache size. We have not experimented with different values of this parameter, nor with variations of the history length 100 used to track fetch times and application process compute times.

Forestall's dependence on batch-size is similar to *aggressive*'s. We used for *forestall* the batch sizes given in table 6.

To compare static vs. dynamic estimation of *forestall*'s parameter F', we compared the performance of *forestall* using fixed values of 1, 2, 4, 8, 15, 30, and 60 for F' to its performance using the dynamic estimation just described. Because actual average inter-reference compute times in our traces vary greatly (from 1.3ms for postgres-join to 15.7ms for the postgres-select), no single value can work well for all traces.

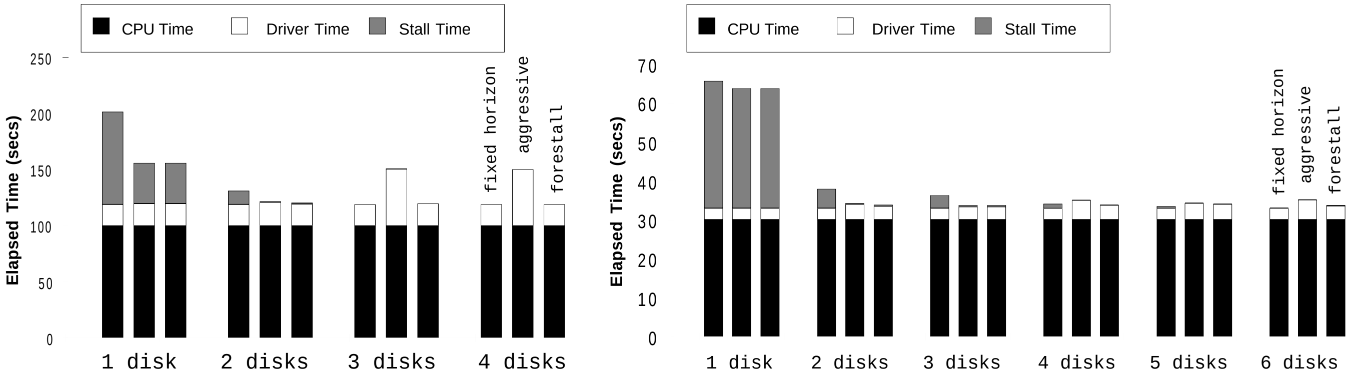


Figure 8: Performance on the synth (left) and xds (right) traces. Each group of bars represents the performance of the three algorithms *fixed horizon*, *aggressive*, and *forestall*, in left-to-right order.

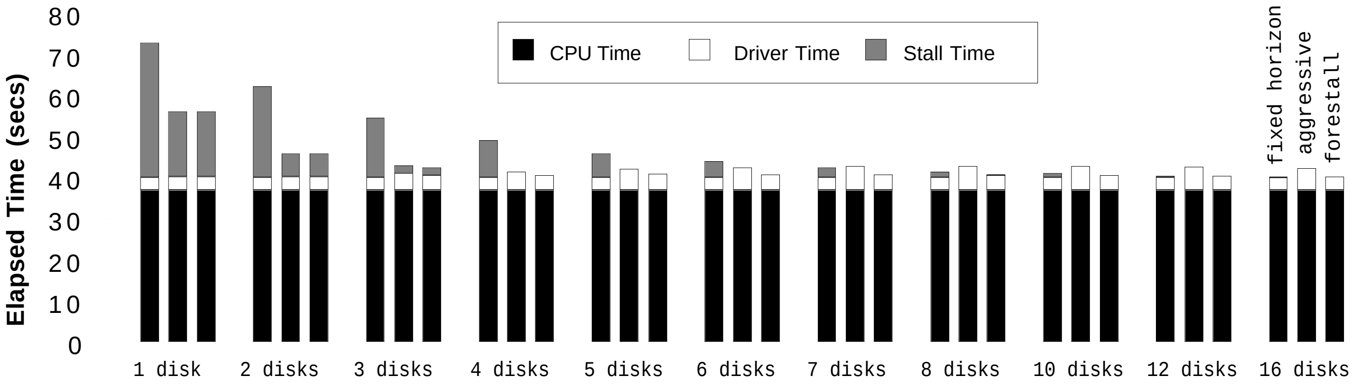


Figure 9: Performance on the cscope2 trace.

The values with the least maximum degradation relative to the dynamic algorithm, over all traces and disk array sizes, are 30 and 60; performance is at most 6.8% worse than the dynamic estimation (on the j2 trace with 2-6 disks). We exclude the synth trace from this calculation, since its artificially low disk response times demand a very low value of F'. For each trace, there is a fixed choice of F that works well across all disk array sizes. This best choice varies from 1 for the dinero trace to 60 for the glimpse trace. The maximum degradation relative to the dynamic estimation allowing this much flexibility is 1.4% (for the ld trace with 7 disks and $F' = 30$). Finally, if we choose the best fixed value for each trace and each disk array size, the maximum degradation relative to the dynamic estimation is 1.2% for the cscope1 trace with one disk and $F' = 2$; for all other traces and array sizes, as well as all other array sizes for this trace, the degradation is less than .5%.

These results indicate that choosing the right parameters between workloads is more important than choosing the right parameter within a particular workload. Furthermore, *forestall*'s performance even with a single fixed parameter over all workloads and array sizes is always within 7% of optimal, and is almost always within 4% of optimal. This suggests that the advantages of *forestall* are due to its estimation of and adaptivity to upcoming disk load rather than the dynamic nature of its fetch-time and compute-time estimates.

5.1 Performance of *forestall*

Figure 8 shows the performance of the three practical algorithms, *fixed horizon*, *aggressive* and *forestall*, on the synthetic trace and xds. *Forestall* behaves exactly as expected. In the I/O bound situations, it prefetches aggressively enough to perform as well as or even better than *aggressive*. In the CPU-bound situations, it becomes more conservative in its prefetching, and has a lower driver overhead, matching the performance of *fixed horizon*.

Figures 9 and 10 show the performance of the three algorithms on the cscope2 and glimpse traces. Once again, *forestall* has the best performance of the three practical algorithms. On all remaining traces, over all configurations, *forestall*'s performance was between 2% worse and 5.8% better than the best of *aggressive* and *fixed horizon* in that configuration. For the full data, see [17].

Table 8 shows the utilization of the disks by *forestall* on the postgres-select trace. Its utilization falls between those of *aggressive* and *fixed horizon*, as expected. Moreover, in I/O-bound situations, it places a load on the disks similar to *aggressive*'s; in compute-bound situations, it places a lower load on the disks, similar to that of *fixed horizon*.

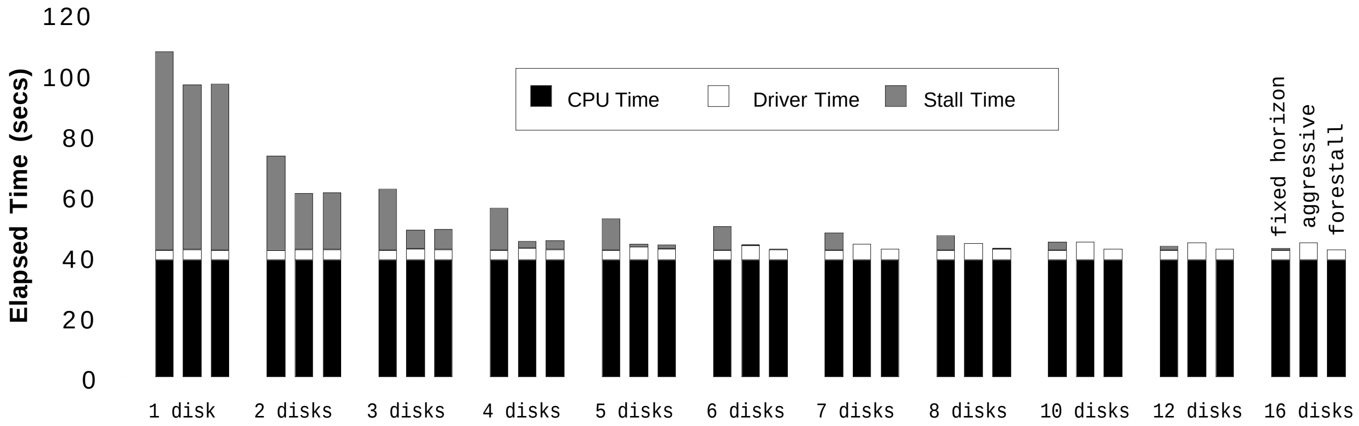


Figure 10: Performance on the glimpse trace.

disks	1	2	3	4	5	6	7	8	10	12	16
util.	.99	.92	.87	.81	.68	.63	.62	.54	.39	.30	.32

Table 8: Utilization of disks by *forestall* on the postgres-select trace.

6 Conclusions

This paper presents the results of a trace-driven simulation study of integrated prefetching and caching algorithms on a single read-only access sequence, assuming that all accesses are known in advance. We studied four algorithms: *aggressive*, *fixed horizon*, *reverse aggressive*, and *forestall*. We found that the theoretically near-optimal *reverse aggressive* usually has the best performance of the four algorithms, but that, perhaps surprisingly, it was never much better than the best of the other algorithms. This shows that carefully choosing replacements is not necessary to balance the load across the disks when the data is well laid out. We found that each of *aggressive* and *fixed horizon* performs well under the conditions for which it was designed, and in any given situation, one or the other performs similarly to *reverse aggressive*. Clearly, *aggressive* and *fixed horizon* are much more practical algorithms than *reverse aggressive*. These observations led us to the hybrid approach of *forestall*, which prefetches more aggressively in I/O-bound situations and more conservatively in compute-bound situations, resulting in nearly the best performance of the four in all configurations.

This study leaves several important issues unresolved. The performance of the algorithms depends on a set of parameters which interact in a complicated way with the applications' access patterns and inter-reference compute times, the layout of data on disks, the disk-scheduling discipline, and the characteristics of the disks. At this time, we have no analytical basis for dynamically determining *aggressive*'s batch size, *fixed horizon*'s prefetch horizon H, *reverse aggressive*'s batch sizes and estimate of F, or *forestall*'s batch size and estimate F' of F. It is a challenging open problem to fully understand the interaction between the algorithmic parameters and the specific application and system characteristics.

Another direction for future research is the treatment of writes, both theoretically and experimentally.

We have not considered the effects of incomplete or inaccurate hints and we have not dealt with the question of how to allocate buffers among competing processes. While the three practical prefetching algorithms can easily be adapted to deal with these situations ([5, 26]), we expect differences in their performance. Aggressive prefetching increases both disk utilization and cache utilization. Therefore, disks are more likely to be busy when unhinted accesses occur. Moreover, an aggressively prefetching process might consume too large a fraction of the cache relative to a nonhinting process. Since *fixed horizon* places the least load on the disks and the cache, it is likely to be least affected by unhinted accesses and to have the smallest impact on other executing processes.

Lastly, this work reaffirms that the operating system can effectively take advantage of hints. An important research direction is to determine how applications can easily provide such hints.

Acknowledgements

Tracy Kimbrel and Anna Karlin wish to thank Martin Tompa for his continued encouragement and advice. We could not have managed the production of this document without the help of Dylan McNamee. Karin Petersen, our paper shepherd, was of great help in improving the quality of presentation.

This research is supported in part by NSF grant numbers ECD-8907068, CCR-9301186, GER-9450075, CCR-9632769, in part by DARPA Contract numbers DABT63-94-C-0049, DABT63-93-C-0054, in part by generous contributions from the member companies of the Parallel Data Consortium, and in part by Intel Corporation and Digital Equipment Corporation. Tracy Kimbrel is supported by an Intel Foundation Graduate Fellowship. Brian Bershad is supported by an NSF Presidential Faculty Fellowship. Ed Felten is supported by an NSF National Young Investigator Award. The views and conclusions contained in this document are those of the au-

thors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz and A. Sussman. Tuning the Performance of I/O-Intensive Parallel Applications. *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, pages 15–27, May, 1996.

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

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

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

[5] 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.

[6] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of Integrated Prefetching and Caching Strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages , May 1995.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] Tracy Kimbrel and Anna R. Karlin. Near-optimal Parallel Prefetching and Caching. In *Proceedings of the 1996 IEEE Symposium on Foundations of Computer Science*, October 1996.

[17] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. Technical Report UW-CSE-96-09-01, University of Washington, 1996.

[18] 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.

[19] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report PCS-TR94-220, Department of Computer Science, Datmouth College, July 1994.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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*, pages 79-95, December 1995.

[27] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modelling. In *IEEE Computer*, 27(3):17-28, March 1994.

[28] K. Salem and H. Garcia-Molina. Disk Striping. In *the 2nd IEEE Conference on Data Engineering*, pages 336–342, Feb. 1986.

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

[30] 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.