Parallel Prefetching and Caching

by

Tracy Kimbrel

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by ______________________________
(Co-Chairperson of Supervisory Committee)

Martin Tompa
(Co-Chairperson of Supervisory Committee)

____

Program Authorized
to Offer Degree Department of Computer Science and Engineering

Date July 24, 1997

UMI Number: 9806992

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature ____________

Date July 24, 1997

University of Washington

Abstract

Parallel Prefetching and Caching

by Tracy Kimbrel

Co-Chairpersons of Supervisory Committee: Professor Anna R. Karlin
Professor Martin Tompa
Department of Computer Science
and Engineering

High-performance I/O systems depend on prefetching and caching 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 may cause a block that is needed soon to be evicted from the cache, thus reducing the effectiveness of the cache, while a block cached too long may reduce the effectiveness of prefetching by denying opportunities to the prefetcher. Using both analytical and experimental methods, we study the problem of integrated prefetching and caching for an I/O system with multiple disks.

In a theoretical analysis, we consider algorithms for integrated prefetching and caching in a model abstracting relevant characteristics of file systems with multiple disks. Previously, the "aggressive" algorithm was shown by Cao, Felten, Karlin, and Li to have near-optimal performance in the single disk case. We show that the natural extension of the aggressive algorithm to the parallel disk case is suboptimal by a factor near the number of disks in the worst case. Our main theoretical result is a new algorithm, "reverse aggressive," with provably near-optimal performance in the presence of multiple disks.

Using disk-accurate trace-driven simulation, we explore the performance characteristics of several algorithms in cases in which applications provide full advance

knowledge of accesses using hints. The algorithms tested are the two mentioned previously, plus the "fixed horizon" algorithm of Patterson *et al.*, and a new algorithm, "forestall," 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.

We also consider three related problems. First, we present an optimal algorithm for a restricted version of the single disk prefetching and caching problem. Next, we propose an approach to the integration of prefetching and caching policies with processor and disk scheduling policies. Finally, we show the NP-hardness of a problem of ordering requests to maximize locality of reference.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

This dissertation presents a study of prefetching and caching strategies for achieving high performance in computer systems. The algorithms developed are evaluated in the context of file systems (i.e., main memory caching of disk-resident data). However, the same principles apply to any prefetching and caching problem. The understanding and some of the techniques developed here could prove useful between any two levels of a memory hierarchy, as well as in any other context in which caching and prefetching can be used to improve performance such as a networked information storage and retrieval system.

1.1 Prefetching and caching

Prefetching and caching are fundamental techniques for achieving high performance at low cost in computer systems. Prefetching and caching exploit the characteristic of most computer applications known as *locality of reference*: a recently used data item is likely to be re-used soon (*temporal locality*), and nearby data items are likely to be used in the near future as well (*spatial locality*).

Caching refers to the technique of storing copies of data that are likely to be used or re-used in the near future in a *cache*, which is a smaller, faster, and thus more expensive (per unit of storage) device than the device that holds them for the long term, which is known as the *backing store*. On a request for a data item, the cache is searched first. If the item is present, we say the request *hits* in the cache; otherwise, the request is a *miss* and the data item must be retrieved from the slower backing store. This technique is used at many levels in a typical computer system: there are one or more levels of processor cache, the processor cache is backed by main memory, main memory is backed by disk storage, and disk storage may be backed by tape storage. This design is known as a *memory hierarchy.* (A processor's register set, smaller and faster than the first level of processor cache, is part of this hierarchy as well, but is not referred to as a cache since it is accessed explicitly.)

If a cache is smaller than the set of data being consumed, then a *replacement strategy* must be used to determine which data item to *evict* from the cache to make room for an incoming item. Some caches, known as *direct-mapped* caches, allow no flexibility in this choice. Each data item is allowed to reside in only a single particular location in the cache, so that whichever item occupies an incoming item's location must be evicted to make room for the incoming item. We will be concerned mostly with *fully-associative* caches in this thesis, in which any data item may occupy any cache location. *Set-associative* caches fall between the other two forms: each data item is allowed to reside in any of a set of cache locations, so that there is a choice of data items to evict, but there is not as much flexibility as there is with a fully-associative cache. A set-associative cache in which each data item maps to a set of size t is termed *t-way-associative.*

Caching is generally effective in overcoming the gap between the *bandwidth* of the backing store, that is, the maximum rate of data transfer, and the rate of data consumption. A large enough cache can eliminate references to the slower backing store to the point that the backing store's bandwidth equals or exceeds the rate at which data are requested from it. Caching also reduces the problem of *latency*, the delay from the time at which a data item is requested to the time at which the item is delivered. However, caching alone is unable to completely overcome latency. *Prefetching* is a technique to hide latency: if a request for data can be issued to the backing store long enough *before* the data are needed, the backing store can return the data to the cache by the time they are needed.

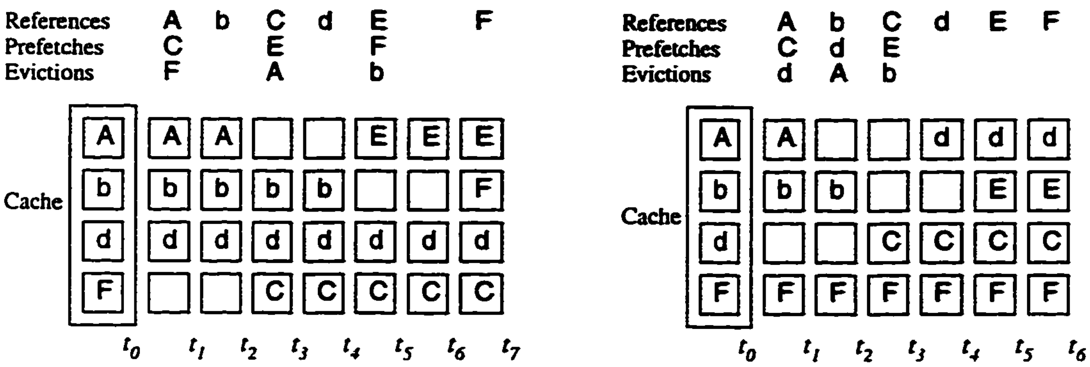
When a data item is prefetched, space in the cache must be allocated. Prefetching displaces data from the cache earlier than would be necessary if data were fetched *on demand*, i.e., if each request were not issued to the backing store until a cache miss occurs. Because of this, more cache misses may occur in a prefetching system than in a demand fetching system. This thesis explores the tension between prefetching and caching and the tradeoffs raised by it.

To prefetch effectively, some form of advance knowledge of future data requests must be available. This knowledge, referred to as *lookahead*, may be complete or incomplete, exact or inexact. Only some of the future requests may be known, or errors may be present in the lookahead information. Many caches perform what can be thought of as a form of prefetching that exploits spatial locality by fetching data in blocks that contain two or more neighboring data items. The only lookahead information used is the "guess" that neighboring data items are likely to be used in the near future. This thesis considers prefetching and caching strategies for cases in which detailed lookahead information is available. The question of the source of lookahead information is orthogonal to the question of how best to use the information for prefetching and caching. This thesis is concerned with the latter question.

1.2 An example

An example will serve to introduce our model and illustrate the challenge posed by the multi-disk problem. An application program 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. Each disk can perform only one fetch at a time. If the cache is full, every fetch requires the eviction of some block from the cache. In a real system, it is not known in advance exactly how long a fetch will take (though in our theoretical model, the fetch time is constant); because of this, we assume 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. In the following example, the cache holds four blocks, and it takes two time units to fetch a block from disk.

Suppose the application references blocks according to the sequence (A, b, C, d, E, F), and the cache initially holds blocks A, b, d, and F. Blocks A, C, E, and F reside on one disk; blocks b and d on a different disk. A straightforward approach is to use the *aggressive* algorithm [7]: 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. Figure 1.1(a) shows the schedule of prefetches, evictions, and block service times produced by this algorithm. For example, initially, the first missing block is



(a) A prefetching and caching schedule.

(b) A better schedule.

Figure 1.1: An example of prefetching and caching with two disks.

C, and the block whose next reference is furthest in the future is F. Moreover, the reference to F is after the reference to C. Thus, the *aggressive* algorithm immediately initiates a fetch for C, evicting F. Notice that this fetch is entirely overlapped with computation (the references to A and b). The schedule produced using this algorithm results in one unit of stall time (the sixth time unit). The entire sequence is served in seven time units.

Figure 1.1(b) shows another schedule that is faster by one time unit. On the first fetch, d is evicted rather than F, even though d is referenced earlier than F. This has the advantage of offloading one fetch from the heavily loaded disk to the otherwise idle disk. This change allows the fetches of C and d, and of d and E, to proceed in parallel, thus saving one time unit.

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

1.3 Applications

The primary application area of the techniques considered in this thesis is that of file systems. A typical modern file system uses a portion of a computer's main memory as a cache (referred to as a *file cache*), backed by one or more disk drives. Advances in technology have made magnetic disks both cheap and small. As a result, parallel disk arrays have become an attractive means for achieving high file system performance at low cost. Multiple disks offer the advantages of both increased bandwidth and reduced contention. Nonetheless, there are many applications which do not benefit from this I/O parallelism as much as they could, and end up stalling for I/O a significant fraction of the time. At the same time, it has been observed that many of these applications have largely predictable access patterns. This has enabled the use of prefetching and informed cache replacement(e.g., [12, 25, 35, 36]) as techniques for reducing I/O overhead in such systems.

Typical disk drive response times are in the 5-30 millisecond range [38]. This large cache miss cost makes it worthwhile to spend a large amount of effort to avoid cache misses. The techniques in this thesis require nontrivial amounts of computation to make available and maintain lookahead information and its relation to the cache state. As processor speeds continue to increase, the increasing miss penalty between a processor's cache and main memory (currently many tens of processor cycles) will make it feasible to expend more resources and use similar techniques at higher levels of the memory hierarchy. A case in point is the problem considered in Chapter 7. In that chapter, a problem related to the main results of this thesis is considered. The results described in Chapters 3–6 address the problem of exploiting locality of reference and lookahead information to maximize performance. The problem considered in Chapter 7 is one of increasing locality of reference by carefully constructing request sequences. This idea has been applied at the level of the processor cache to avoid main memory accesses [37].

1.4 Relation to previous work

Caching and prefetching have been known techniques to improve performance of storage hierarchies for many years [4, 13]. In computer architecture, work on caching and prefetching has focused on bridging the performance gap between processors and main memory [41]. Research using caching and prefetching in database systems [10, 33, 11] 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 [13, 28, 20, 31, 42, 5, 18, 6, 36]. The most common prefetching approach is to perform sequential readahead, i.e., to detect when an application accesses a file sequentially, and to prefetch in order the blocks of the files that are so used [13, 28, 29]. The limitation of this approach is that it benefits only applications that make sequential references to large files. Recently, caching and prefetching have also been studied for parallel file systems [12, 25, 35].

Another body of work has been on predicting future access patterns [12, 42, 33, 11, 18]. A recent trend is to use applications' knowledge about their access patterns to perform more effective caching and prefetching [5, 6, 35, 36]. Patterson *et al.* [35] describe a mechanism by which an application's request sequence can be made known in advance. They use a *hinting* interface through which an application can be explicitly programmed to diclose its future file accesses to the file system. Mowry *et al.* [30] use a different mechanism to make an application's demands known to the system. There, compiler techniques are applied to regularly-structured computations to predict applications' virtual memory page faults. An advantage of their mechanism is that the lookahead information is generated automatically, without any effort on the part of the application developer. A disadvantage is that applicability of the method is limited to applications with access patterns that are amenable to prediction through compiler analysis.

Much research on parallel I/O has concentrated on techniques for striping and distributing error-correction codes among redundant disk arrays or other devices. These techniques are used to achieve high bandwidth by exploiting parallelism and to tolerate failures [21, 39, 9, 34, 17]. For purposes of this thesis, *striping* will refer to a data layout in which block i of a file resides on disk ($i \bmod d$), where d is the number of disks. This allows any d consecutive blocks to be fetched in parallel, thus benefiting programs that exhibit spatial locality.

The work presented in this thesis complements these previous efforts. File access prediction or application-provided lookahead information can be used to provide the inputs to the algorithms considered here. The *reverse aggressive* algorithm, described in Chapter 2, has provably near-optimal performance for any given mapping of disk blocks to disks. Its performance will only improve when a near-optimal layout is used. As described in Chapter 4, a striped layout allows algorithms that are simpler and more practical to compete with the provably near-optimal algorithm *reverse aggressive*.

1.4.1 The integrated prefetching and caching problem

Our problem is a generalization of, but much more complicated than, the classical paging problem. Indeed, one principle for prefetching (the *optimal eviction* rule described in Section 2.3) is derived from Belady's optimal *longest forward distance* paging algorithm [4]. As we will see, however, the application of this rule alone is insufficient to guarantee good prefetching performance.

We know of no prior theoretical analysis of the integration of prefetching and caching in the presence of multiple disks. There have been some interesting results on the use of data compression for the design of optimal prefetching strategies [27, 44], and work on prefetching strategies for external merging under a probabilistic model of request sequences [32]. However, these studies concentrated only on the problem of determining which blocks to fetch, and did not address the problem of determining which blocks to replace.

This thesis builds on recent studies by Cao, Felten, Karlin, and Li of the single-disk prefetching and caching problem [7, 8]. They showed that it is important to integrate prefetching, caching and disk scheduling and that a properly integrated strategy can perform much better than a naive strategy, both theoretically and in practice. Cao *et al.* proposed the *aggressive* prefetching and caching algorithm, which is described in detail in Chapter 2. Another closely related line of research is the work of Patterson, Tomkins, Gibson, Ginting, Stodolski, and Zalenka [35, 36, 43]. Patterson *et al.* proposed the *fixed horizon* prefetching and caching algorithm [36]. *Fixed horizon* is described in detail in Chapter 2. Mowry *et al.* [30] do not separate the generation of lookahead information from its use, as do the other works mentioned. Their compiler inserts prefetch requests in the code it generates. These are placed to request each block a fixed amount of time ahead of the cache miss, much as the *fixed horizon* algorithm of Patterson *et al.* does. The lookahead information generated by their compiler could be used by any of the prefetching and caching algorithms considered in this thesis.

As mentioned in Section 1.1, the generation of lookahead information is orthogonal to, albeit necessary for, its use by a prefetching and caching algorithm. This thesis focuses on the use and not the generation of lookahead information. We present the *reverse aggressive* algorithm (developed in joint work with Anna Karlin) and the *forestall* algorithm (developed in joint work with Tomkins, Patterson, Bershad, Cao, Felten, Gibson, Karlin, and Li). Like *aggressive* and *fixed horizon*, these algorithms address the integrated prefetching and caching problem for a single process. All four can be modified to deal with incomplete and inexact lookahead information, as discussed in Chapter 2.

Aggressive and *fixed horizon* were designed under different workload assumptions. *Aggressive* was designed assuming a single disk, which is expected to be a performance bottleneck. Thus, *aggressive* maximizes utilization of this constrained resource, which is appropriate. *Fixed horizon* was designed assuming enough I/O parallelism so that each request is issued to an idle disk. *Fixed horizon* prefetches more conservatively than *aggressive* to make the best cache replacement for each prefetch. It does this by delaying each prefetch until there is just enough time to complete it in time for the reference. Again, this is appropriate under the workload for which the algorithm was designed; however, it causes problems when the assumption of abundant disk bandwidth is violated. As we will see in Chapter 4, each of these algorithms performs well under the conditions for which it was designed, but each suffers under the workload for which the other is designed.

In contrast to *aggressive* and *fixed horizon*, *reverse aggressive* and *forestall* were designed to take advantage of any amount of I/O parallelism. *Reverse aggressive* carefully constructs a schedule of prefetches and evictions while considering the loads placed on the multiple disks. This ensures that the loads are balanced. *Reverse aggressive* is able to do this for any layout of data. All three other algorithms suffer from a load imbalance problem in the worst case. *Forestall* achieves a compromise between the aggressive prefetching of *aggressive* and the conservative prefetching of *fixed horizon*. It does this by estimating the time at which prefetching must be initiated to avoid causing the application process to *stall*, i.e., to wait for a prefetch to complete because it was not initiated soon enough. This differs from *fixed horizon*'s mechanism in that more than one prefetch is considered at a time when deciding how early to begin prefetching. This allows *forestall* to deal with congestion, i.e., the situation in which more than one block must be prefetched from the same disk. With a striped data layout, *forestall* is able to maintain load balance without the careful deliberation of *reverse aggressive*. (*Aggressive* and *fixed horizon* are able to do so as well.) In practice, striping eliminates the load imbalance problem of worst-case data layouts.

The algorithms described above determine prefetching and caching schedules for a single application process. Cao *et al.* and Patterson *et al.* propose different policies to allocate cache resources to multiple, competing processes. Cao *et al.* [5, 6, 8] describe *LRU-SP*, which determines cache allocations based on those that would be obtained using the *least recently used (LRU)* replacement policy, applied globally to all processes' interleaved reference streams. The *cost-benefit* analysis of Patterson *et al.* [36] compares the cost of one process giving up a cache buffer to the benefit of reallocating that buffer to another process. These are measured in terms of the time saved or lost by a process divided by the amount of time the buffer is used or given up. The choice of prefetching and caching algorithm is orthogonal to that of the cache allocator, as well as to that of the lookahead generation mechanism.

Along with the hinting interface of Patterson *et al.* or the compiler-generated lookahead method of Mowry *et al.*, each of the algorithms for prefetching and caching represents a complete solution to the problem of improving I/O performance for those applications that are amenable to the chosen lookahead generation mechanism. When a hint-generating tool and a prefetching and caching algorithm are combined with one of the cache allocation mechanisms described above, a complete solution for multi-programmed workloads is obtained. Tomkins *et al.* have gone on to evaluate *forestall* experimentally in conjunction with each of the *cost-benefit* and *LRU-SP* allocation mechanisms, as well as *aggressive* in conjunction with *LRU-SP* and *fixed horizon* in conjunction with *cost-benefit* [43] .

In addition to the single-process parallel prefetching and caching problem, this thesis also addresses three related issues. The first is the development of an efficient algorithm to find optimal prefetching and caching schedules. We make partial progress by finding such an algorithm for a restricted version of the single disk prefetching and caching problem. The second is the integration of prefetching and caching policies with processor and disk scheduling policies. With this idea, we take a wider view than the previous studies of integrated prefetching and caching for multiple processes, which assumed standard time-sharing scheduling mechanisms. The third is the ordering of request sequences for increased locality of reference to be exploited by prefetching and caching systems.

1.5 Contributions and organization of thesis

This dissertation makes the following research contributions:

- A theoretical understanding of the performance benefit that is made possible through effective prefetching and caching techniques for a system with multiple backing stores is presented.

- A theoretical understanding of techniques that achieve the possible benefit is presented.

- A practical algorithm that achieves the aforementioned performance benefit in the presence of complete and accurate application-provided advance knowledge of file system data requests is developed and evaluated.

- An efficient optimal algorithm for a restricted version of the single disk prefetching and caching problem is given.

- A step is made toward an understanding of the interaction of prefetching and caching strategies with processor and disk scheduling policies and a technique for integrating them is proposed.

- A partial analysis of a problem of scheduling independent threads of control to increase locality of reference and thereby improve cache performance is given.

This dissertation is organized as follows. Chapter 2 describes file prefetching and caching in greater detail and presents a framework that is common to the problems considered in Chapters 3– 6. Chapter 2 also describes several algorithms for the parallel prefetching and caching problem. Chapter 3 contains a theoretical treatment of the parallel prefetching and caching problem. Previously proposed algorithms are analyzed, and a new algorithm, *reverse aggressive*, is shown to have near-optimal performance for the abstract parallel prefetching and caching problem. *Reverse aggressive* is not a practical algorithm. However, it serves as a benchmark against which to compare practical algorithms. Perhaps more important, an understanding of the reasons for *reverse aggressive*'s good performance leads to the design of a more practical algorithm that is able to match its performance. Chapter 4 describes an experimental evaluation of algorithms for prefetching and caching in a file system with multiple disks. Traces of file accesses by real applications are used to drive a simulator, which gives accurate estimates of the performance characteristics of the algorithms considered. A practical algorithm, *forestall*, is found to match or exceed the performance of *reverse aggressive* in trace-driven simulations. Chapter 5 proposes a new approach to the single disk prefetching and caching problem. An efficient algorithm is given to find a schedule that does not stall for any sequence for which such a schedule exists. Chapter 6 proposes a strategy for the integration of prefetching and caching policies with processor and disk scheduling for a multi-programmed workload. An algorithm is described that finds an optimal solution to a simplified version of the problem. It is argued that in conjunction with *forestall*, the algorithm may nonetheless provide a practical solution to a problem that appears to be very difficult to analyze. Chapter 7 describes a mechanism that has been proposed previously by which performance can be improved by increasing a program's locality of reference [37]. The problem of finding an optimal solution is shown to be NP-hard. Chapter 8 summarizes the thesis and presents conclusions and directions for future study.

Preliminary versions of the results presented in Chapter 3 were presented in [23]. The results presented in Chapter 4 appeared previously in [24].

Chapter 2

THE PARALLEL PREFETCHING AND CACHING PROBLEM

In this chapter, we describe a theoretical model that captures the important characteristics of a system for prefetching and caching with multiple backing stores. We also describe several prefetching and caching algorithms. Because the primary motivation for this problem comes from file systems, we will refer to the backing stores as *disks*. In Chapter 3, we study the offline problem of constructing an optimal prefetching and caching schedule in this model, for a given stream of requests for blocks of data residing on the disks. An optimal schedule minimizes the elapsed time required to serve a given request stream.[1] Although complete information about future requests is usually not available, partial information *is* often available in the form of limited or even significant lookahead into the request stream. All the algorithms considered here can be modified to deal with inexact and incomplete lookahead, as described in Section 2.5. In addition, the design and analysis of an optimal offline algorithm is an important step towards understanding and evaluating more practical limited-lookahead algorithms. We can perhaps draw an analogy with the impact of the optimal offline paging algorithm [4] on the design, implementation and evaluation of online paging algorithms [40].

Our model is read-only. The algorithms we consider can improve the performance of read-only and read-mostly applications (i.e., those for which the performance impact of write traffic is negligible). An interesting open problem is the integration of the algorithms considered here with techniques to improve write performance.

Surprisingly, even in the read-only, complete and accurate lookahead, single-disk situation, this is a challenging combinatorial problem. We know of no polynomial time algorithm for determining an optimal prefetching schedule. The difficulty comes

[1] In this chapter, we consider only the case of a single request sequence. Chapter 6 discusses performance measures appropriate for a multi-programmed workload.

from the fact that prefetching too soon can cause additional cache misses by replacing blocks that would remain in the cache if prefetching were done later or not at all: new and possibly better eviction opportunities arise as a program proceeds. Nonetheless, Cao et al. [7] were able to show that a simple and natural algorithm called *aggressive*, which prefetches as early as is reasonable, has performance that is provably close to optimal in the single disk case.

We show in Chapter 3, however, that the natural extension of this algorithm to the multiple disk case has performance that is suboptimal by a factor nearly equal to the number of disks. The interaction between caching and prefetching is significantly more complicated in a system with multiple disks because a set of blocks can be prefetched in parallel only if they reside on different disks: each disk can serve only one prefetch at a time. The prefetching schedule and choice of cache evictions impact the potential for subsequent parallel prefetching in a complex way. Our main theoretical result is a new algorithm, *reverse aggressive*, with provably near-optimal performance for this problem.

2.1 Theoretical model

Our model generalizes the example of Section 1.2 in the obvious way.

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

We imagine that there is a *cursor* which at any time points to the next request to be served. If this request is for a block that is in the cache, the cursor advances by one during the next time unit. If this request is for a block that is not in the cache, the cursor *stalls* until that block arrives in the cache (i.e., until the fetch for that block completes). Note that to the extent that the cursor is advancing, a prefetch can overlap the serving of requests. Also, prefetches can overlap each other provided that the prefetched blocks reside on different disks. We assume that each block resides on only a single disk.

The goal is to determine a schedule of prefetches and evictions such that the time required to serve the entire sequence is minimized. Since it requires one unit of time to serve each request, the elapsed time is equal to the length of the request sequence plus the total number of steps during which the cursor stalls.

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

2.2 Algorithms for parallel prefetching and caching

We consider five algorithms for parallel prefetching and caching in this thesis, *conservative*, *aggressive*, *reverse aggressive*, *fixed horizon*, and *forestall*. The first two are natural extensions of the two single disk prefetching strategies described by Cao *et al.* [7]. They lie at opposite ends of the spectrum in terms of the total number of fetches performed: *Conservative* performs the minimum possible number of fetches, at the expense of a worse elapsed time in the worst case; *Aggressive* prefetches as aggressively as possible without being foolish. *Fixed horizon* and *forestall* lie between these extremes. For all four of these algorithms, there are reference patterns on which their performance is suboptimal by a factor of nearly d, for values of d, F and K that are typical in practice.

Our main theoretical result is the development and analysis of a new algorithm, called *reverse aggressive*, whose performance is provably close to optimal. Interestingly, it achieves this by constructing a prefetching schedule backwards, i.e., by considering the reference sequence in reverse order. For reasons that will be made clear, this causes it to avoid problems encountered by the (forward) *aggressive* algorithm. *Aggressive* suffers from load imbalance and an inability to keep lightly loaded disks from outpacing (prefetching far ahead of) heavily loaded disks. We give an intuitive explanation of *reverse aggressive*'s advantages in Section 2.4.3. Detailed proofs are contained in Chapter 3.

2.3 Prefetching with a single disk

Before proceeding, we review the results of Cao et al. [7] for prefetching and caching in the single-disk case. They described four properties that can be assumed of any optimal strategy in the single-disk case. These constrain the problem and by adhering to them, we can ensure that an algorithm's performance is not far from optimal.

1. *optimal fetching*: when fetching, always fetch the missing block that will be referenced soonest;

2. *optimal eviction*: when fetching, always evict the block in the cache whose next reference is furthest in the future;

3. *do no harm*: never evict block A to fetch block B when A's next reference is before B's next reference;

4. *first opportunity*: never evict A to fetch B when the same thing could have been done one time unit earlier.

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

Nonetheless, these rules do provide a fair amount of guidance in the design of a prefetching algorithm. Cao et al. considered two natural algorithms, *aggressive* and *conservative*, that follow these rules and lie at opposite ends of the spectrum of possibilities. *Aggressive* is the algorithm that initiates a prefetch whenever its disk is ready (i.e., is not in the middle of a prefetch) and the *do no harm* rule allows it. *Conservative* is the algorithm that refuses to fetch until it can evict the block that would be evicted by Belady's optimal *longest forward distance* [4] algorithm in the classical paging model. Belady's algorithm suffers the fewest page faults among all paging algorithms. It does this by evicting the page not needed for the longest time among all blocks in the cache whenever the next request is missing from the cache. *Conservative* makes the minimum number of total fetches, but it often declines opportunities to prefetch blocks.

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

2.4 Detailed descriptions of algorithms for parallel prefetching and caching

There is an obvious and natural extension of each of *conservative* and *aggressive* to the multi-disk case.

---

[2] F corresponds to the ratio of disk access time to the application program's inter-access time. Our traced applications described in Chapter 4 spend over one millisecond computing between requests, on average, and average disk response times are under 20 milliseconds, yielding $F \leq 20$. A 10 megabyte cache holds 1280 8-kilobyte blocks. Putting these together, we have $F/K \leq 0.016$.

2.4.1 Conservative

(Multi-disk) conservative is the following algorithm: Construct a page replacement schedule using the *longest forward distance* offline paging algorithm. For each fetch/eviction pair in this schedule, initiate a prefetch as soon as the evicted page is referenced for the last time (until the schedule specifies that it is to be fetched back into the cache) and the disk containing the fetched page is free (i.e., the previous prefetch from that disk is complete).

Conservative applies the rule *optimal eviction* as though the prefetch were to be initiated immediately before serving the request to the missing block, then applies the rule *first opportunity* (perhaps many times) to swap the chosen fetch/eviction pair as early as possible.

We will see in Chapter 3 that *conservative*'s performance can be suboptimal by a factor greater than the number of disks in the worst case.

2.4.2 Aggressive

(Multi-disk) aggressive is the following algorithm: Whenever a disk is free, prefetch the first missing block on that disk, replacing the block whose next reference is furthest in the future, provided this does not violate *do no harm*.

Aggressive is the most aggressive prefetching strategy that is consistent with the four optimal prefetching rules described in Section 2.3. As we shall see in Chapter 3, *aggressive* does not enjoy the same performance guarantee in the multi-disk case as it achieved in the single disk case. In fact, the four properties on which it was based in the single disk case do not hold for optimal strategies in the multi-disk case. As a result, it suffers from two problems in the multi-disk case that did not exist in the single disk case:

- The eviction decisions it makes are "color-blind." It chooses evictions to make without consideration of the load on the disks. These choices can result in a situation where many of the holes at any time are of the same color, and therefore can not subsequently be prefetched in parallel. (See Figure 1.1 for an example of this.)

- *Aggressive* is too aggressive. The result is that it can cause some disks to fetch too far ahead with respect to other disks. These fetches increase the share of the cache occupied by blocks belonging to the lightly loaded disk(s), creating even more holes for the heavily loaded disk(s) to fill.

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

2.4.3 Reverse aggressive

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

Optimal eviction is more troublesome. Suppose there are two disks, colored red and blue. If there are many red blocks missing in the sequence, say, it may be that the best choice for eviction is a blue block even though the block whose next request is furthest in the future is red. This is because the relatively lightly-loaded blue disk can better handle the increased burden of another missing block than the red disk can. (See Figure 1.1.) Given that a blue block is to be evicted, say, it is true that the best choice is the blue block that is not requested for the longest time. That is, the colored version of this rule holds, but it does not tell us which color block to evict.

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

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

First, we return to the single disk case, and observe that any prefetching schedule that serves the reverse sequence S^r in time T can be used to derive a schedule to serve S in time T as follows. If the schedule for serving S^r serves request r_i between times t and $t+1$, the derived schedule for S serves r_i between times $T-t-1$ and $T-t$. If the reverse schedule replaces a with b between times t and $t+F$, the derived schedule replaces b with a between times $T-t-F$ and $T-t$.[3] Applying this logic twice, we see that the optimal elapsed time for the reverse sequence is the same as the optimal elapsed time for the original sequence.

Reversal of the sequence is more complicated when multiple disks are considered. In the forward direction, the prefetching schedule is constrained to fetch at most one block at a time from each disk; eviction choices may be blocks of any color. Switching between the forward sequence and the reverse sequence, fetches become evictions and vice versa. To derive a useful schedule from a schedule serving the reverse sequence, then, requires that the schedule for the reverse sequence be constrained to *evict* at most one block of each color at a time. This is illustrated in the following example (see Figure 2.1):

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

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

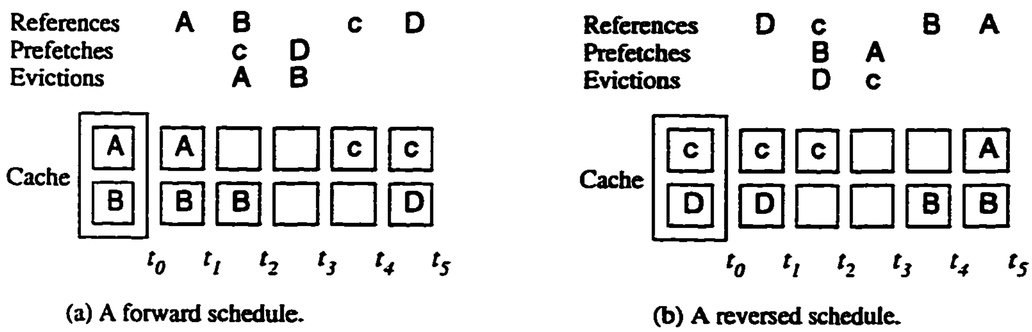


Figure 2.1: An example of reversing a schedule of prefetching and caching with two disks.

blue disk, evicting A. At time 2, a fetch of D from the red disk is initiated, evicting B from the cache. The schedule serves the request sequence in five units of time. See part (a) of Figure 2.1.

In the schedule for the reverse sequence, at time 1, D is evicted to start fetching B. Since c is blue and D is red, a fetch of A (evicting c) can be started at time 2, even though A and B are both red. See part (b) of Figure 2.1. This schedule can be transformed as described in the previous paragraphs into the valid schedule for the forward sequence of part (a), which is its mirror image.

As previously mentioned, all of the rules presented in Section 2.3 except *optimal eviction* can be assumed of optimal prefetching schedules for the reverse sequence. *This fact makes it easier to find a schedule for the reverse sequence, then transform it into one for the original sequence, than to find a schedule for the original sequence directly.* The reason for this is that in the forward direction, any time a block is prefetched a decision must be made as to which color block to evict. In the reverse direction, this decision is made for us: the block to evict is the one not needed for the longest time whose color matches the color of the free disk. (I.e., the "colored" version of the *optimal eviction* rule can be used.) One might expect that fetch decisions are harder, but this is not the case. In the forward direction, the missing block to fetch is the one of the right color that is needed soonest. (This is the colored version of *optimal fetching* described earlier.) In the reverse direction, it is the one needed soonest, regardless of color.

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

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

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

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

2.4.4 Fixed horizon

Fixed horizon is the following algorithm: Whenever there is a missing block at most F references in the future and the disk on which it resides is free, issue a fetch for that block, replacing the cached block whose next reference is furthest in the future, provided that the *do no harm* rule is satified (which will certainly be true if $F < K$).

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 performs well. 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 F 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 would expect *fixed horizon* to have performance much closer to optimal than its worst case. We will see in Chapter 3 that in the worst case, *fixed horizon* can be suboptimal by a factor nearly equal to the number of disks.

2.4.5 Forestall

Aggressive and *fixed horizon* are simpler than *reverse aggressive* and more practical. As we will see in Chapter 4, despite their worst case lower bounds, they perform well under a striped layout of data, since many real programs exhibit spatial locality. However, each has a weakness. *Aggressive* prefetches too aggressively in compute-bound situations, and *fixed horizon* prefetches too conservatively in I/O-bound situations. A highly-tuned version of *reverse aggressive* is able to perform comparably to the best of *aggressive* and *fixed horizon* in any situation, but it is not a practical algorithm: it is more complicated, and it requires a complete reverse pass over its input before the application can begin processing its data.

Forestall is an algorithm designed to combine the best features of all the 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.

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. Further suppose that the subsequent two requests are missing from the cache. *Aggressive* immediately starts fetching and avoids stalling on the two holes, 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 hole; it then stalls $F-1$ steps on the second hole. In contrast, suppose there is only one hole 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 hole, 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} hole in the request sequence. For any $i \geq 1$, if $iF > d_i$, processing will surely stall on the i^{th} hole or some earlier hole. 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 is the following algorithm: Whenever a disk is free, let d_i denote the distance from the cursor to the i^{th} missing block that resides on the disk. If $d_i \leq iF$ for any $i < K$ and the *do no harm* rule allows a prefetch, prefetch the first missing block that resides on the disk, evicting the block that is not used for the longest time among all blocks in the cache.

Notice that *forestall* achieves an effect similar to one achieved by *reverse aggressive*. Each fetch is completed just in time to avoid a stall, *subject to the need to fetch more than one block from the same disk.* This is the second of the two advantages of *reverse aggressive* over *aggressive* mentioned in Section 2.4.3. The first, the ability to maintain balanced loads on the disks, can be achieved by using a striped data layout.

2.5 Coping with imperfect lookahead

In practice, lookahead information may not be perfect. Several forms of incomplete and inexact lookahead are possible. It may be merely *limited.* That is, complete and accurate lookahead may be available for some number of future requests, but no more. Limited lookahead information is likely to arrive in "chunks." The application provides a chunk of lookahead, then proceeds to consume the corresponding data. Some requests may be missing from the lookahead information. This could happen when the entire lookahead sequence is available in advance, or when it is limited. Another possibility is that the lookahead contains blocks in a different order than that in which they are subsequently requested. Finally, the lookahead sequence may contain blocks that are never requested by the application. If this occurs too frequently, it may be best to ignore the lookahead and fetch data on demand instead. An interesting model (not considered in this thesis) is one in which the lookahead information contains estimates of the probability of its correctness.

As mentioned, the algorithms can be modified to deal with imperfect lookahead information. This is rather straightforward for all the algorithms except *reverse aggressive*. The other algorithms can simply prefetch using whatever lookahead is available, behaving as they would if it were the real reference sequence. If all the lookahead is used up at any time, no more prefetching occurs until more lookahead becomes available. When a block is requested that is missing from the cache because it was not present in the lookahead information or because the request comes earlier in the sequence than indicated by the lookahead, a fetch should be started as soon as the disk is free. If there is not enough lookahead information to determine which block is not needed for the longest time among all blocks in the cache, the algorithms can fall back on LRU replacement (i.e., replace the least recently used block in the cache among those that are not present in the yet-to-be-consumed lookahead). Implementations of *aggressive*, *fixed horizon*, and *forestall* that incorporate these features have been developed by Patterson, Tomkins, *et al.* [36, 43].

Modifying *reverse aggressive* is more complicated. Before describing an implementation that copes with imperfect lookahead, we describe a simple trick needed to implement the algorithm even in the case of perfect lookahead. We can assume without loss of generality that any (forward) schedule for a request sequence leaves the last K distinct requests in the cache, since no evictions are necessary once these are present in the cache. The "natural" *reverse aggressive* algorithm leaves the last K requests of the reverse sequence in the cache at the end of its reverse schedule. These are the first K distinct requests in the forward sequence. However, if we assume the cache is empty at the time the application starts, we must somehow load it with the first K distinct requests. The simple trick is to append requests for K dummy blocks to the end of the reverse sequence before running *reverse aggressive*. To "serve" these requests, *reverse aggressive* flushes all real blocks from its cache on the reverse pass. This sequence of evictions becomes a schedule of prefetches to load the cache with the first K distinct requests in the forward direction.

Now consider the limited lookahead case. For the first chunk of lookahead information, the scheduling problem is the same as in the complete lookahead case: the cache is initially empty, and contains the last K distinct requests in the chunk at the end of the schedule. (If there are fewer than K requests in the chunk, the cache will contain all the blocks in the chunk, and will also "contain" some of the dummy blocks.) We can think of this as producing a schedule that starts with the cache full of dummy blocks, and ends with the cache containing some other set of blocks. For subsequent lookahead chunks, the problem is to produce a schedule starting with the cache contents at the end of the schedule for the previous chunk, rather than with the cache full of dummy blocks. Thus we need to append the current cache contents to the (reversed) lookahead chunk before running *reverse aggressive*. An implementation of this approach was developed during the early stages of the simulations reported in Chapter 4 [22].

The modifications described to make the other algorithms deal with other forms of imperfect lookahead can be applied to *reverse aggressive*, now that we have a mechanism for producing a schedule given some amount of partial lookahead information.

Chapter 3

THEORETICAL ANALYSIS

3.1 Overview of results

This chapter presents the results of joint work with Anna Karlin [23]. All the algorithms are shown to perform nearly d times worse than optimal in the worst case, with the exception of *reverse aggressive*. *Reverse aggressive* is shown to perform within $1 + F/K$ of optimal in the worst case.

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

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

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

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

Theorem 3 *The previous lower bound applies to* fixed horizon *and* forestall *for $d \ll \sqrt{F}$: There are arbitrarily long strings on which each requires time $d - \frac{3d(d-1)}{F+3(d-1)}$ times the optimal elapsed time (within an additive constant that depends only on F and K).*

Theorem 4 Reverse aggressive *requires at most* $1 + dF/K$ *times the optimal elapsed time to service any request sequence, plus an additive term* dF *independent of the length of the sequence.*

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

On real systems, dF/K is small[1], so that the factor $1 + dF/K$ in Theorem 4 is not much greater than one (hence our claim of "near-optimality").

In the remainder of this section we give high-level descriptions of the main ideas used to derive our results. Full details are given in Section 3.2.

3.1.1 Performance of conservative, aggressive, fixed horizon, *and* forestall

The key concept in the upper bound of Theorem 2 is the notion of *domination* from the work on prefetching and caching in the single-disk case [7]. This allows us to bound the cost of *aggressive*'s prefetching schedule in terms of the progress of the optimal schedule at intermediate points during the processing of the request sequence.

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

Let *opt* denote an optimal algorithm. For intuition, consider the following. If *aggressive*'s cursor is ahead of *opt*'s cursor, *aggressive*'s holes dominate *opt*'s holes, and both are initiating prefetches at the same times, then *opt*'s cursor cannot pass *aggressive*'s: while *aggressive* stalls on a hole, *opt*'s cursor cannot pass its matching hole. We show that *aggressive* is able to continually regain and maintain such an advantage (having its cursor ahead and its holes dominate) over *opt* at regular intervals, without losing too much time to *opt* in the process. *Aggressive* can lose its

---

[1] F/K is typically less than 0.02 as described in Chapter 2. Small disk arrays with at most 5-10 disks are most common in practicem and are likely to continue to be so [15, 45]. Moreover, technological trends are such that F/K will only get smaller with time: disk and memory speeds (which dominate F) change slowly, while memory size increases exponentially [19].

advantage, and lose time to *opt*, by prefetching more aggressively than *opt*; this will become clear as the details are presented.

The lower bounds of nearly d in Theorems 1, 2, and 3 come from the fact that an adversary can construct request sequences that cause the algorithms to always fetch blocks from only one disk (because they make poor eviction choices). The optimal algorithm *opt* can serve these same sequences at nearly d times the rate because of the parallelism of prefetching on d disks. The additive term of one for *conservative* (in both the upper and lower bounds) comes from *opt*'s ability to overlap prefetches with the serving of requests. In contrast, *conservative* may not be able to do so.

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

3.1.2 *Performance of* reverse aggressive

The proof of Theorem 4 required several new ideas. The notion of domination from the proof of Theorem 2 is replaced by a stronger notion that we call *strong domination*.

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

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

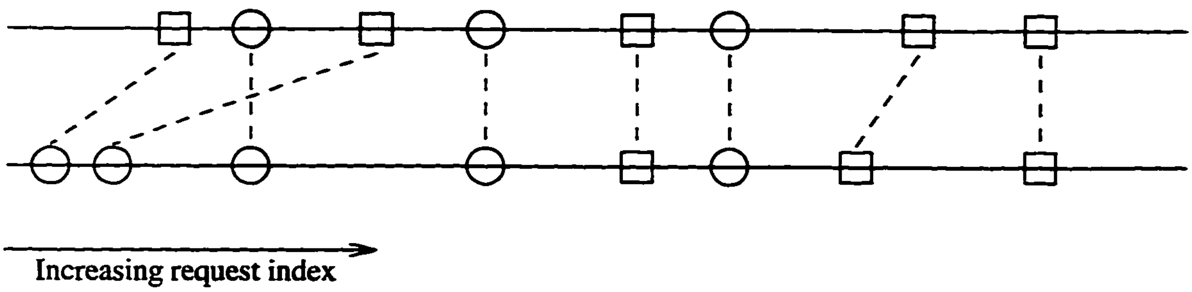


Figure 3.1: Strong domination example: the upper set of holes strongly dominates the lower one.

This idea is illustrated in Figure 3.1, in which holes of different colors are depicted by different shapes. Mismatched shapes represent excess holes. Edges show how strong domination implies ordinary domination. (See Lemma 7 and the discussion following.)

Definition: For two sets A and B of holes, we say that A strongly dominates B up to index y, if the subset of holes in A that occur at or before index y in the request sequence strongly dominates the subset of holes in B that occur at or before index y. When y is the end of the request sequence, we will simply use "strongly dominates" rather than "strongly dominates up to the end of the sequence."

Definition: Let $New(H, (c, color))$ denote the new set of holes should a prefetch be initiated, if possible (i.e., if allowed by the *do no harm* principle), evicting a block of color *color*, when the cursor position is c and the current set of holes is H. Note that $New(H, (c, color))$ is uniquely determined by the optimal prefetching principles *optimal fetching* and *colored optimal eviction* described in Section 2.4.3. If the *do no harm* principle prevents a prefetch, define $New(H, (c, color)) = H$.

The following crucial lemma is used to show that if *reverse aggressive* strongly dominates *opt*, and both have the opportunity to initiate a fetch replacing blocks of the same color, then *reverse aggressive* strongly dominates *opt* after the corresponding fetches complete.[2] For purposes of analysis, we consider any blocks that are currently

[2] We are speaking here of the performance of *reverse aggressive* on the reverse sequence, compared to an optimal schedule for the reverse sequence. However, as described in Section 2.4.3, the optimal elapsed time is the same in both directions, and from *reverse aggressive*'s schedule, we are able to derive a prefetching schedule for the forward sequence with the same elapsed time.

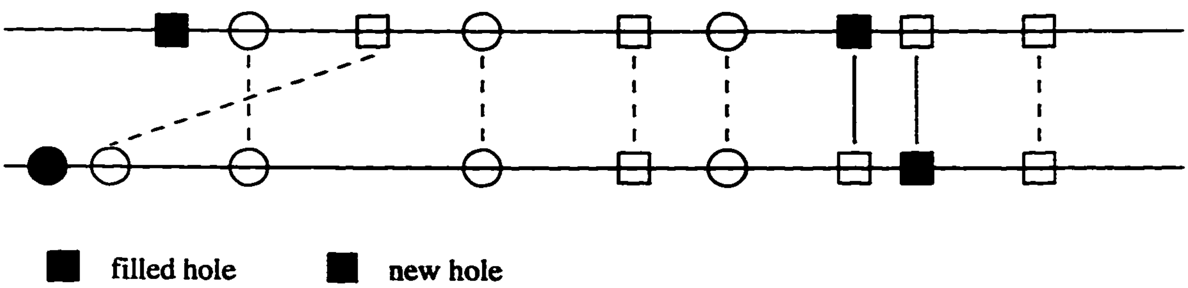


Figure 3.2: Domination lemma: the upper set of holes continues to strongly dominate the lower one.

being fetched to be in the cache, i.e., there is no corresponding hole in A or B, even though the corresponding request cannot be served until the F steps are over.

Lemma 5 Strong Domination Lemma

Let A and B be two sets of holes in a request sequence R, and let y, $c_A < y$, and $c_B < y$ be indices in R. If A strongly dominates B up to index y, then:

1. *For each color col, if $c_A \geq c_B$, $New(A, (c_A, col))$ strongly dominates $New(B, (c_B, col))$ up to y.*

2. *For each color col, $New(A, (c_A, col))$ strongly dominates B up to y.*

3. *For each color col, if $c_A \geq c_B$ and every block of color col that is* not *a hole in A is requested after c_A and before the first hole in A so that $New(A, (c_A, col)) = A$ (i.e.,* do no harm *prevents a prefetch), A strongly dominates $New(B, (c_B, col))$ up to y.*

4. *For each pair col_A and col_B of colors, if the best eviction choice of color col_A given the set of holes A and the cursor position c_A is a block that is not requested between c_A and y, $New(A, (c_A, col_A))$ strongly dominates $New(B, (c_B, col_B))$ up to y.*

Part 1 of Lemma 5 is illustrated in Figure 3.2.

Note that part 3 of Lemma 5 is a special case of part 1. We prove it separately because it is an important case and because it will aid understanding later, where the lemma is used.

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

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

Formalizing these arguments is difficult; the details are presented in Section 3.2.

3.2 Proofs

3.2.1 Terminology

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

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

If algorithm a has fetches in progress at any time t, we denote a's holes before initiating those fetches by $H_a^-(t)$ (i.e., H_a^- contains the holes being filled, but not the ones being created), and a's holes after those fetches complete (but ignoring any fetches that haven't begun by time t) by $H_a^+(t)$.

In this section and the next, we assume all algorithms are working with the reverse sequence, and denote the optimal algorithm for serving the reverse sequence by *opt*.

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

A fetch using the busy disk (and evicting a block of the same color as the busy disk; the block fetched may be any color) is referred to as a *busy-disk fetch*; fetches using other disks are referred to as *non-busy-disk fetches.*

3.2.2 Reverse aggressive*: upper bound*

Outline of the proof

We first give some preliminaries, proving the claims of Section 2.4.3 and a simple lemma on combining subsets of dominating and dominated sets of holes. We next prove the strong domination lemma (Lemma 5).

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

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

Detailed proof

Lemma 6 *Any prefetching schedule that does not satisfy the four rules described in Section 2.4.3 can be transformed into one that does, with no increase in elapsed time.*

Proof:

1. *optimal fetching* (fill the first hole): Suppose that at time t_1, a fetch is initiated to fill some hole h_2 other than the first hole h_1. h_1 must be filled before it can be served; say it is filled by a fetch initiated at time $t_2 > t_1$. Since the (later) reference to h_2 cannot be served until after the reference to h_1 is served, the schedule remains valid if h_1 is filled at time t_1 and h_2 at time t_2, and all other operations (prefetches, evictions, and cursor movements) are unchanged. Since we are working with the reverse sequence, this change can be made regardless of the colors of h_1 and h_2. If filling h_1 at time t_1 allows the cursor to advance sooner than it can if h_2 is filled at time t_1, then the eviction opportunities under this schedule are at least as good as those under the original schedule; i.e., the set of holes obtained strongly dominates that obtained under the original schedule. Thus the transformed schedule can be completed to derive a schedule with elapsed time no greater than that of the original.

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

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

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

□

We assume without loss of generality that *opt* obeys these rules.

Lemma 7 *Given two sets of holes $A = A_1 \cup A_2$ and $B = B_1 \cup B_2$ with $|A_1| \leq |B_1|$, $|A_2| \leq |B_2|$, $A_1 \cap A_2 = \emptyset$, and $B_1 \cap B_2 = \emptyset$, if A_1 dominates B_1 and A_2 dominates B_2, then A dominates B.*

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

dominates B_2, for each member of A there is a distinct member of B with lesser or equal index. Thus we have a contradiction. □

Note that Lemma 7 extends to pairs of sets composed of more than two disjoint subsets each. Notice also that by Lemma 7, strong domination implies (ordinary, color-blind) domination. (Match non-excess holes according to colors, and all of one set's excess holes to all of the other set's excess holes. See Figure 3.1.)

Lemma 8 *Strong domination is transitive.*

Proof: Suppose A strongly dominates B and B strongly dominates C. We show that A strongly dominates C. Fix a color c; for convenience (so it can be used as an adjective), suppose c is red. Define $N_c(\cdot)$ as before. For a collection S of sets of holes, let $N_c(S) = \min_{s \in S}(N_c(s))$. (We will drop the brackets when listing the members of S.) Let $N_c = N_c(A, B, C)$. We consider three cases, illustrated in Figure 3.3.

1. $N_c = N_c(A)$. A has N_c red holes, and these dominate the last N_c red holes in B. B's last $N_c(B, C)$ red holes dominate C's last $N_c(B, C)$ red holes, so B's last N_c red holes must dominate C's last N_c red holes. Since domination is transitive, A's N_c red holes dominate C's last N_c red holes. Suppose h is a red hole in C that is excess with respect to A. If h is matched to a red hole h' of B, h' is excess with respect to A and thus precedes A's first hole, so h must precede A's first hole as well. If h is excess with respect to B, it precedes B's first hole, which precedes or is the same as A's first hole, since strong domination implies ordinary domination.

2. $N_c = N_c(B)$. A's last N_c red holes dominate B's N_c red holes, which dominate C's last N_c red holes. Suppose h is a red hole in C that is excess with respect to B. h must precede B's first hole. h precedes A's first hole as well, since B's first hole precedes or is the same as A's first hole; again, this is because strong domination implies ordinary domination. If h is excess with respect to A, we are done. If h matches some hole h' of A, h surely does not occur after h'.

3. $N_c = N_c(C)$. A's last $N_c(A, B)$ red holes dominate B's last $N_c(A, B)$ red holes, so A's last N_c red holes must dominate B's last N_c red holes, which dominate C's N_c red holes. C has no excess red holes with respect to B or A.

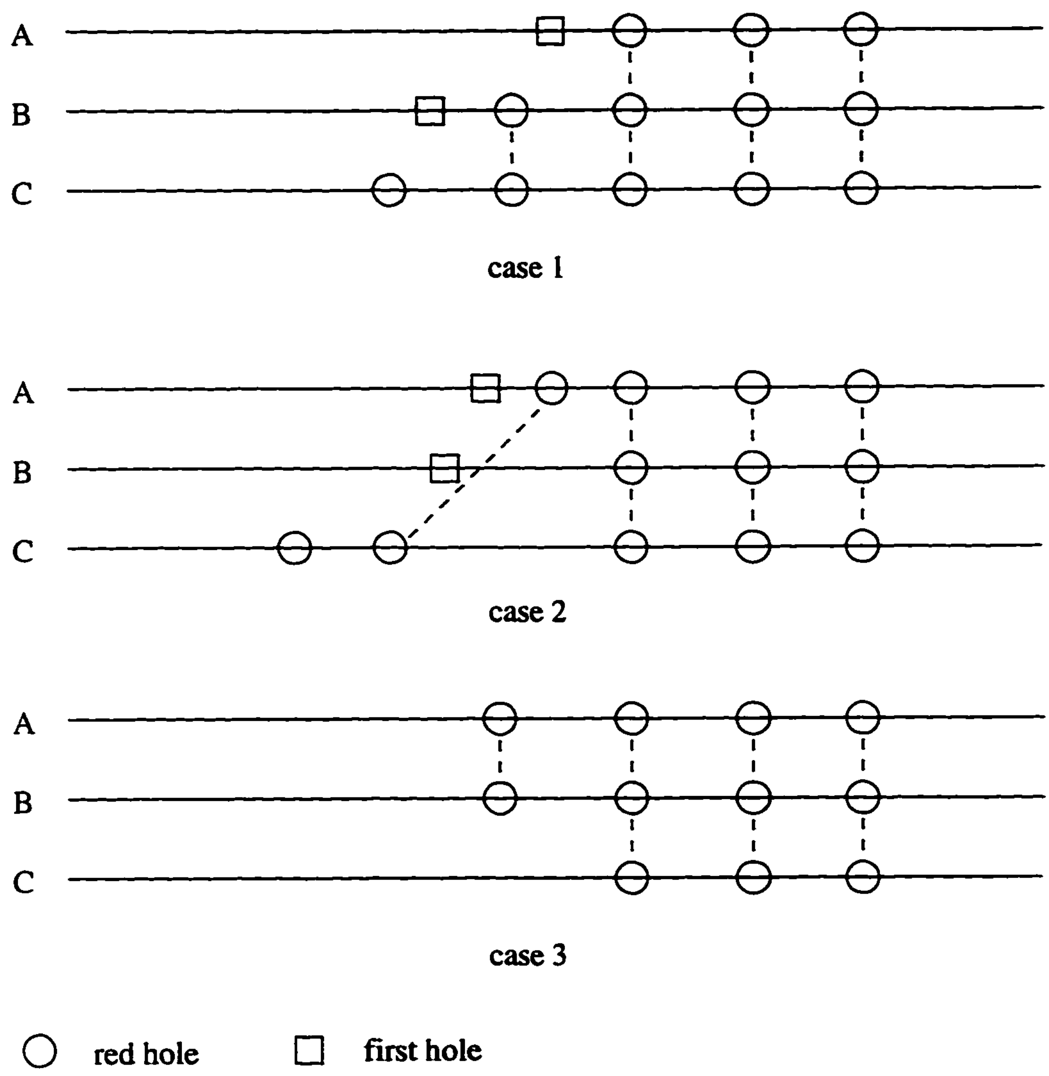


Figure 3.3: Strong domination is transitive.

□

We now prove Lemma 5 (the Strong Domination Lemma).

Proof: Define $N_c(A)$, $N_c(B)$, and N_c as before.

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

1. A's first hole is removed (if necessary).
2. B's new hole is added to B (if necessary) and A's new hole is added to A (if necessary).
3. B's first hole is removed (if necessary).

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

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

First we prove part 1.

Step 1: A's first hole is filled

Let c be the hole's color. First, since A's new first hole is to the right of its old first hole (the one being filled), B's excess holes all are still to the left of A's first hole. If c was an excess color of A, we are done. Otherwise, B's hole that was matched to A's filled hole becomes an excess hole, and since it occurred no later than the hole it matched, it is to the left of A's new first hole. Notice that $|A| < |B|$ at this point, in addition to the fact that A strongly dominates B.

Step 2: eviction

If $c_A > y$, A's new hole does not affect strong domination up to y, and the addition of a new hole to B (whether left of, right of, or at y) cannot affect strong domination. If $c_A \leq y$, let A's last N_c holes of the same color c as the block evicted occur at indices $a_1 < a_2 < \ldots < a_{N_c}$, and let B's occur at $b_1 < b_2 < \ldots < b_{N_c}$. Since A strongly dominates B, we know that $a_i \geq b_i$ for each i. Let B's new hole be its j^{th} non-excess hole of color c, i.e., the new hole occurs between b_{j-1} and b_j, or at an index greater than b_{N_c} in which case $j = N_c + 1$, or before b_1 in which case $j = 1$. (As a special case, if c is an excess color of B, and the new hole is left of B's last excess hole of color c, the new hole becomes an excess hole and the last excess hole takes its place in the following argument.) Let A's new hole be its r^{th} hole of color c, with a special case similar to that in the definition of j. Let $a'_1 < a'_2 < \ldots < a'_{N_c+1}$ be the indices of A's last $N_c + 1$ holes of color c after the eviction, and let $b'_1 < b'_2 < \ldots < b'_{N_c+1}$ be the indices of B's last $N_c + 1$ holes of color c after the eviction. Then for $i < r$, $a'_i = a_i$ and for $i > r$, $a'_i = a_{i-1}$; for $i < j$, $b'_i = b_i$ and for $i > j$, $b'_i = b_{i-1}$. To show that domination is preserved, we need to show that $a'_i \geq b'_i$ for each i, $1 \leq i \leq N_c + 1$. For $i < \min(r, j)$ and $i > \max(r, j)$ it is immediate that $a'_i \geq b'_i$. If $r > j$, then we have

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

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

$$\begin{aligned} a'_j &\geq b'_j \\ a'_{j-1} &\geq b'_{j-1} \\ &\cdots \\ a'_{r+1} &\geq b'_{r+1} \\ a'_r &\geq b'_r. \end{aligned}$$

Suppose that one or more of these inequalities does not hold, and let i be the largest index for which $a'_i < b'_i$. Then either $i = j = N_c + 1$ and $a'_i < b'_i$, or

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

where A's new hole at a'_r satisfies $a'_r \leq a'_i$. In either case, there is a block that is not requested until index b'_i that is not a hole in A, and the new hole in A is a block requested earlier at index a'_r instead. But the definition of *New* states that the best possible eviction choice is made, i.e., that the block evicted is the block whose next occurrence is at the greatest index among all blocks of color c in the cache. Thus we have a contradiction.

Since the holes of color other than c are unaffected by this change, and domination of holes of color c is preserved, strong domination is preserved. Also, we still have that $|A| < |B|$.

Step 3: B's first hole is filled

Let c be the hole's color. If c is an excess color of B, then B will have one fewer excess hole of color c; the remaining ones are unchanged, and thus are still to the left of A's first hole. Otherwise, the hole was matched to some hole of A, which becomes an excess hole. The newly excess hole's position is relevant in the definition of strong domination only if it is A's first hole; in this case, since neither A's first hole nor B's excess holes are changed, strong domination is preserved. Because $|A| < |B|$ before this step, we have that $|A| \leq |B|$ afterwards, as needed for strong domination.

The proof of part 1 is complete.

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

For part 3 nothing happens in step 1. Let c be the color of B's new hole. Again, for step 2, an argument similar to that for part 1 shows that A's non-excess holes of color c dominate B's non-excess holes of color c; if not, A would contain a hole to the left of the next request for some block that is not a hole. If c is not an excess color of B, we are done with step 2. Otherwise, we need to show that all of B's excess holes of color c precede A's first hole. Suppose that B has $N_c + 1$ holes of color c at or to the right of A's first hole. A has only N_c holes of color c, so B has some hole h of color c that is not a hole of A and is to the right of A's first hole. Again, A would then contain a hole to the left of the next request for some block that is not a hole. Step 3 is the same as for part 1.

The proof of part 4 is an easy simplification of that of part 1, since A's new hole is beyond y and need not be considered. (B's new hole may be beyond y as well.) □

A particular case in which part 3 of Lemma 5 applies deserves mention. It may be that all blocks of some color c are holes in A, i.e., there are no blocks of color c in the cache, and that a fetch is not possible since there is no block of color c in the cache to evict, but that there are blocks of color c that are not holes in B. It may seem that a schedule with B as its set of holes has an advantage since it can make use of its disk c, and a schedule with A as its set of holes can not. But there is no advantage, provided that the conditions of strong domination are met. A is a superior state, and the schedule filling one of the holes in B by evicting a block of color c is merely "catching up" to the other schedule by eliminating one of its excess holes.

Here is our main result.

Theorem 9 Reverse aggressive *requires less than* $1 + dF/K$ *times the optimal elapsed time to service any request sequence, plus an additive term* dF *independent of the length of the sequence.*

Proof: For $d = 1$, the theorem follows directly from the result of [7]. Thus we may assume $d \geq 2$.

We show that for each $i \geq 0$ (numbering the phases starting with 0), there are times T_i and T_i', such that

- T_i' is the time *opt*'s cursor reaches the i^{th} phase;
- *reverse aggressive*'s cursor position at time T_i is at least as great as *opt*'s cursor position at time T_i';
- For $i > 0$, $T_i - T_{i-1} \leq T_i' - T_{i-1}' + dF - 1$.
- $H_{rev}^+(T_i)$ strongly dominates $H_{opt}^+(T_i')$;
- if *reverse aggressive*'s busy disk for phase i will become free (i.e., complete any fetch in progress) in $z \leq F - 1$ steps after T_i, then *opt*'s corresponding disk will not become free until $z' \geq z$ steps after T_i'.

If there are p phases, we take T_p (respectively, T'_p) to be the time at which *reverse aggressive* (respectively, *opt*) finishes serving the request sequence.

The theorem will follow from the first three conditions, as follows. For each phase i, *reverse aggressive*'s elapsed time $e_{rev}(i) = T_{i+1} - T_i$ and *opt*'s elapsed time $e_{opt}(i) = T'_{i+1} - T'_i$ satisfy

$$e_{rev}(i) \le e_{opt}(i) + dF - 1$$

so that

$$\frac{e_{rev}(i)}{e_{opt}(i)} \le 1 + \frac{dF-1}{e_{opt}(i)}.$$

Each phase except possibly the last is of length at least K, so that $e_{opt}(i) \ge K$. Putting these together, we have that for all phases but the last,

$$\frac{e_{rev}(i)}{e_{opt}(i)} \le 1 + \frac{dF-1}{K}.$$

The last phase may be incomplete, i.e., may contain requests for fewer than K distinct blocks. *Reverse aggressive* requires at most $dF - 1$ steps more than *opt* to serve the last phase.

We prove the claims about T_i and T'_i by induction. For the base case ($i = 0$), we take $T_0 = T'_0 = 0$. The fact that the claims hold at this time is trivial. For the inductive step, assume the claims hold for the i^{th} phase. We show that they hold for the $(i+1)$-st phase via a two step process.

- We first show in Lemma 10 that in phase i, *reverse aggressive* (starting at time T_i) loses at most $(d-1)F$ steps to *opt* (starting at time T'_i).

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

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

Lemma 10 *Suppose that at time T_i,* reverse aggressive*'s cursor is at position p_i in the sequence. Let $T'_i + t_O(j)$ (respectively, $T_i + t_R(j)$) denote the time at which* opt *(respectively,* reverse aggressive*) serves the request at cursor position $j \geq p_i$, for any j such that r_j is in phase i. Then for all j in the phase, $t_R(j) \leq t_O(j) + (d-1)F$.*

Proof: For the sake of contradiction, suppose the contrary, and consider the least index ℓ such that $t_R(\ell) > t_O(\ell) + (d-1)F$.

First, consider the case in which ℓ precedes the first hole in $H^+_{rev}(T_i)$. Each of *reverse aggressive*'s fetches in progress at time T_i completes by time $T_i + F - 1$, so that *reverse aggressive*'s cursor cannot stall more than $F-1$ steps before reaching the first hole in $H^+_{rev}(T_i)$. Recall we have assumed $d \geq 2$; thus we have a contradiction.

The remainder of the proof of Lemma 10 (and the bulk of that of Theorem 9) consists of the remaining case, in which ℓ is at or beyond the first hole in $H^+_{rev}(T_i)$. By the minimality of ℓ, $t_R(\ell-1) \leq t_O(\ell-1) + (d-1)F$, and *reverse aggressive* stalls at least one step more than *opt* on request r_ℓ. In particular, *reverse aggressive* stalls at time $T_i + t_R(\ell) - 1$, and *opt* does not stall at time $T'_i + t_O(\ell)$. *Reverse aggressive* initiates a prefetch for the block requested at index ℓ at time $T_i + t_R(\ell) - 1 - x$ for some $0 \leq x \leq F-1$; at this time, r_ℓ is *reverse aggressive*'s first hole. We will show that *opt* must have a hole that it has not yet begun to fill at an index no greater than ℓ at time $T'_i + t_O(\ell) - x$, and thus cannot serve r_ℓ before time $T'_i + t_O(\ell) - x + F > T'_i + t_O(\ell)$. Recall that *reverse aggressive*'s busy disk will be free in $z \leq F - 1$ steps after T_i, and *opt*'s corresponding disk will be free in $z' \geq z$ steps after T'_i.

Reverse aggressive will perform busy-disk fetches continuously, initiating a fetch at time $T_i + z + bF$ for each $b \geq 0$, at least until such a time as there are no holes left in the phase. Once there are no holes left in the phase, *reverse aggressive* will not stall at least until the end of the phase is reached. Let b and δ be such that $t_R(\ell) - 1 - x - z = bF + \delta$ and $0 \leq \delta < F$. Then *reverse aggressive* has filled b holes by busy-disk fetches by time $T_i + t_R(\ell) - 1 - x$, and *opt* has filled at most $b - d + 1$ holes by busy-disk fetches by time $T'_i + t_O(\ell) - x$, since

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

Let n be the number of non-busy-disk fetches initiated by *opt* by time $T'_i + t_O(\ell) - x$.

Consider the sequence $S = ((c_1, color_1), \ldots, (c_{n+b-d+1}, color_{n+b-d+1}))$ of fetches *opt* initiates after time T'_i and at or before time $T'_i + t_O(\ell) - x - F$, where the pair $(c, color)$ denotes that a fetch evicting a block of color *color* is initiated at cursor position c. For each fetch $(c', color')$ of *opt*, we define a *matching fetch opportunity* of *reverse aggressive*. A matching fetch opportunity is a pair $(c, color)$ such that *reverse aggressive* has the opportunity to initiate a fetch of color *color* at a cursor position at least as great as c. Each matching fetch opportunity to a fetch in S allows *reverse aggressive* to initiate a fetch (if allowed by the *do no harm* principle) by time $T_i + t_R(\ell) - 1 - x - F$. They are defined as follows:

- Let *opt*'s j^{th} non-busy-disk fetch be initiated at time $T'_i + t'_j$. This fetch is matched to the fetch on the same disk that *reverse aggressive* initiates (if any) in the time interval

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

  Note that by the minimality of ℓ, at time $T_i + t'_j + (d-1)F$ *reverse aggressive*'s cursor is already at or beyond the cursor position at which *opt* initiates its j^{th} non-busy-disk fetch, and its disk of the same color becomes free (finishes any fetch already in progress) within another $F - 1$ steps. Therefore, such a fetch opportunity exists. The fact that *reverse aggressive*'s cursor position at the time of this matching fetch opportunity is at least as great as *opt*'s at the time of its fetch will allow us to apply part 1 or part 3 of the strong domination lemma (Lemma 5) to this pair.

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

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

- *opt*'s j^{th} busy-disk fetch is matched to the j^{th} busy-disk fetch *reverse aggressive* performs in the phase. Since *reverse aggressive* prefetches continuously using its busy disk, we know that each of these fetch opportunities corresponds to an actual fetch. Part 4 of the strong domination lemma will be applied to this pair of fetches.

- Finally, each non-busy-disk fetch initiated by *opt* between times $T_i' + t_O(\ell) - x - F + 1$ and $T_i' + t_O(\ell) - x$ is matched to one of the last $d-1$ busy-disk fetches initiated by *reverse aggressive*. Note that there can be only one such fetch of each color. Part 4 of the strong domination lemma will be applied to this pair of fetches.

We claim that *reverse aggressive*'s holes after these $n + b - d + 1$ matching fetch opportunities pass strongly dominate *opt*'s holes up to the end of the phase after *opt* initiates its sequence S of n non-busy-disk fetches and at most $b - d + 1$ busy-disk fetches. Let R_0 be *reverse aggressive*'s set of holes $H^+_{rev}(T_i)$ at time T_i. Let O_0 be *opt*'s set of holes $H^+_{opt}(T_i')$ at time T_i'. Define O_j, $j \geq 1$, inductively as the set of holes resulting from initiating *opt*'s j^{th} fetch $(c_j, color_j)$ with the set of holes O_{j-1}; i.e., $O_j = New(O_{j-1}, (c_j, color_j))$. Similarly, define R_j, $j \geq 1$, inductively by $R_j = New(R_{j-1}, (c_j, color_j))$. $R_{n+b-d+1}$ is the state that would be reached by starting in *reverse aggressive*'s state R_0, but then initiating fetches (when allowed by *do no harm*) according to *opt*'s prefetching schedule. By a sequence of applications of part 1 and part 3, as appropriate, of the strong domination lemma (Lemma 5), we have that $R_{n+b-d+1}$ strongly dominates $O_{n+b-d+1}$ up to the end of the phase.

We now show that *reverse aggressive*'s holes after its matching fetch opportunities pass strongly dominate $R_{n+b-d+1}$ up to the end of the phase. Because strong domination is transitive (Lemma 8), we will obtain that *reverse aggressive*'s holes strongly dominate *opt*'s up to the end of the phase. Since *opt* and *reverse aggressive* may perform fetches on different disks at different times and in different orders, we need to somehow permute *opt*'s schedule of fetches into *reverse aggressive*'s; then we will be able to make pairwise comparisons between the two sequences of fetches and apply the strong domination lemma. Toward this end, we define the following:

Definition: Consider a fetch sequence, defined by a sequence of triples of the form $(t_j, c_j, color_j)$, where for each j, $t_j \leq t_{j+1}$ and $c_j \leq c_{j+1}$. $(t_j, c_j, color_j)$ denotes a fetch, or an opportunity to fetch, beginning at time t_j with the cursor at position

c_j, where the color of the evicted block is $color_j$. A fetch opportunity denotes an opportunity to fetch in the sense that the disk is free, but no fetch may be possible under the optimal prefetching rules.

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

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

We extend the notation $New(A, (c, color))$ to allow a series of fetches or fetch opportunities, with or without the time indices (which have no effect on the resulting set of holes), in the obvious way: $New(A, S) = New(New(A, f_1), f_2, \ldots, f_{|S|})$ where $S = f_1, \ldots, f_{|S|}$ is a sequence of fetches or fetch opportunites.

Before we can complete the proof of Lemma 10, we need the following three lemmas.

Lemma 11 *Suppose that fetch sequence S within phase i is obtained from fetch sequence S' by a busy-early swap. Then $New(R_0, S)$ strongly dominates $New(R_0, S')$ up to the end of the phase.*

Proof: Let blue denote the color of *reverse aggressive*'s busy disk, and let red denote the color of the first disk to fetch under S' in the swapped pair. We refer to fetches using the blue disk as blue fetches, even though blue is the color of the evicted block; the block fetched may be any color. We refer to fetches using the red disk as red fetches, even though red is the color of the evicted item. The sets of holes of the two sequences immediately before initiating the swapped pair of fetches are the same. In both cases, a blue fetch can be initiated, since by hypothesis there are still holes in the phase. This blue fetch will not require an eviction that creates a new hole within the phase.

Unless the first hole filled is a red block, the set of red blocks in the cache at the time the red fetch is initiated is the same under S' and S. If the first hole is red, then under S', this red block is brought into the cache by the red fetch, and under S, by the blue fetch. Thus, the best eviction opportunity at the time of the red fetch under S is at least as good as that under S', since under S the red fetch occurs at a cursor position c_{j+1} at least as great as that under S', which is c_j.

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

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

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

□

Lemma 12 *Suppose that fetch sequence S is obtained from fetch sequence S' by an overlapping swap. Then for any set A of holes, $New(A, S)$ strongly dominates $New(A, S')$ up to the end of the entire sequence and thus up to the end of the phase.*

Proof: Neither fetch affects the eviction opportunities of the other, since they overlap and evict to different disks. Because they overlap, the first does not bring a block into the cache in time for it to be served before the second fetch starts. An easy consequence of the rules described in Section 2.4.3 is that each block fetched is served at least once before it is subsequently evicted. Because they evict to different disks, the first does not evict a block that could otherwise be evicted by the second.

For each of the two fetches under S', the fetch of the same color under S is initiated at a cursor position at least as great. An argument similar to the proof of Lemma 11 finishes the proof. □

Lemma 13 Reverse aggressive*'s sequence of fetch opportunities can be obtained from the sequence leading to $R_{n+b-d+1}$ (i.e.,* opt*'s sequence of fetches) via a sequence of busy-early swaps, overlapping swaps that do not involve fetches performed by the busy disk, substitutions of busy-disk fetches for non-busy-disk fetches, and insertions of extra fetches not matched to any fetch of* opt.

Proof: The definition of matching fetch opportunities identifies a sufficient set of fetch opportunities. We will show that no operations other than those described are necessary to transform *opt*'s sequence of fetches to *reverse aggressive*'s sequence of matching fetch opportunities.

First we show that for each disk other than the busy disk, any inversion of fetches on that disk and the busy disk is in the "right direction" (i.e., corresponds to a busy-early swap). Let blue denote the color of the busy disk, and let red denote the color of some other disk. For $1 \leq j \leq b$, let $T_i + t_{B_j}$ be the time at which *reverse aggressive*'s j^{th} blue fetch is initiated, and for $1 \leq j \leq b - d + 1$, let $T'_i + t'_{B_j}$ be the time at which *opt*'s j^{th} blue fetch is initiated. For $1 \leq j \leq r$, let t'_{R_j} be the time at which *opt*'s j^{th} red fetch is initiated, and for $1 \leq j \leq r-1$, let t_{R_j} be the time at which *reverse aggressive*'s matching fetch is initiated, where r is the number of red fetches initiated by *opt* at or before $T'_i + T_O(\ell)$.

First, consider all of *reverse aggressive*'s blue and red fetches except its last $d-1$ blue fetches, and all of *opt*'s blue and red fetches except its last red fetch (which is matched to one of *reverse aggressive*'s last $d-1$ blue fetches). We have that for all $j \leq b-d+1$, $t_{B_j} \leq t'_{B_j}$ (i.e., *reverse aggressive*'s j^{th} blue fetch is no later than *opt*'s, by the definition of matching fetch opportunities) and for all $j \leq r-1$, $t_{R_j} \geq t'_{R_j}$ (i.e., *reverse aggressive*'s j^{th} red fetch is no earlier than *opt*'s). Suppose that there is an inversion in the "wrong direction," i.e., that for some j and some k, $t'_{B_j} < t'_{R_k}$ and $t_{R_k} < t_{B_j}$. Then

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

which contains the contradiction $t'_{B_j} < t'_{B_j}$.

Next, consider *opt*'s last (r^{th}) red fetch. Recall that this fetch is matched to one of *reverse aggressive*'s last $d-1$ blue fetches. This requires the substitution of a blue fetch for a red fetch, and possibly some number of busy-early swaps to move the blue fetch forward to its place in *reverse aggressive*'s sequence of fetches; no other red fetches in the sequence are affected by this.

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

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

We now complete the proof of Lemma 10 using Lemmas 11, 12, and 13. We show that *reverse aggressive*'s holes at time $T_i + t_R(\ell) - 1 - x$ strongly dominate *opt*'s holes at time $T'_i + t_O(\ell) - x$ up to the end of the phase, as follows. Let $S_{opt} = S_1, S_2, \ldots, S_m = S_{rev}$ be the series of fetch sequences obtained in the transformation of *opt*'s fetch sequence into *reverse aggressive*'s that was shown to exist by Lemma 13. Recall that we have already shown that $New(R_0, S_{opt}) = R_{n+b-d+1}$ strongly dominates *opt*'s set of holes $New(O_0, S') = O_{n+b-d+1}$ up to the end of the phase. For each $1 < i \leq m$,

$New(R_0, S_i)$ strongly dominates $New(R_0, S_{i-1})$ by Lemma 11, if S_i is derived from S_{i-1} by a busy-early swap, by Lemma 12, if S_i is derived from S_{i-1} by an overlapping swap, by part 2 of the strong domination lemma (Lemma 5), if S_i is derived from S_{i-1} by an insertion, or by part 4 of the strong domination lemma (Lemma 5), if S_i is derived from S_{i-1} by the substitution of a busy-disk fetch for a non-busy-disk fetch. By transitivity of strong domination (Lemma 8), $New(R_0, S_{rev})$ strongly dominates $New(O_0, S_{opt})$ up to the end of the phase.

Thus we have

Corollary 14 Reverse aggressive*'s first hole at time* $T_i + t_R(\ell) - 1 - x$ *is at a cursor position at least as great as* opt*'s first hole at time* $T_i' + t_O(\ell) - x$.

This contradicts the hypothesis that *reverse aggressive* stalls at time $T_i + t_R(\ell) - 1$ and *opt* does not stall at time $T_i' + t_O(\ell)$, and completes the proof of Lemma 10. □

We now use Lemma 10 to complete the inductive step of Theorem 9.

Let T_{i+1}' be the time at which *opt*'s cursor first reaches phase $i+1$ (i.e., one greater than the time at which *opt* serves the last request in phase i). Let f_j' be the j^{th} fetch *opt* initiates after time T_i' and at or before time T_{i+1}', and suppose it begins at time $T_i' + t_j'$. Define the j^{th} *dominating fetch opportunity* to be the fetch opportunity (possibly an actual fetch) that *reverse aggressive* has on the same disk as f_j' in the time interval

$$[T_i + t_j' + (d-1)F, T_i + t_j' + dF - 1],$$

say at time $T_i + t_j$. (Notice this is a different matching than that used in Lemma 10. In this matching, fetches of all colors are matched in the same way non-busy-disk fetches were matched in Lemma 10.) By Lemma 10, we know that *reverse aggressive*'s cursor position at time $T_i + t_j$ is at least as great as *opt*'s cursor position at time $T_i' + t_j'$.

By the same argument as in the proof of Lemma 13, *reverse aggressive*'s sequence of dominating fetch opportunities can be obtained from *opt*'s sequence of fetches by a series of overlapping swaps and insertions. Applying the strong domination lemma (Lemma 5), Lemma 12, and transitivity of strong domination (Lemma 8) as needed, we obtain that *reverse aggressive*'s holes after its dominating fetch opportunities have passed strongly dominate *opt*'s holes after completing its sequence of fetches. This is the same argument as in the proof of Lemma 10, but without the complication of busy-early swaps.

By Lemma 10, *reverse aggressive*'s cursor reaches phase $i+1$ by time $T_i + (T'_{i+1} - T'_i) + (d-1)F$. Within another $F-1$ steps, *reverse aggressive* initiates its dominating fetches matching the ones *opt* has in progress at time T'_{i+1}. A fetch of *opt* started at time $T'_{i+1} - x$ is matched (if needed) by *reverse aggressive* by time $T_i + (T'_{i+1} - T'_i) + dF - 1 - x$; in particular, if *opt* has a fetch in progress on *reverse aggressive*'s busy disk for phase $i+1$ at time T'_{i+1}, that fetch has at least as many steps remaining at time T'_{i+1} as *reverse aggressive*'s fetch (if any) has remaining at time $T_i + (T'_{i+1} - T'_i) + dF - 1$. Thus if we take time T_{i+1} to be $T_i + (T'_{i+1} - T'_i) + dF - 1$, the invariants are restored.

□

3.2.3 Reverse aggressive*: lower bound*

We have been unable to strengthen the lower bound of Cao, Felten, Karlin, and Li [7], which showed that *aggressive* can perform $(1 + (F-1)/K)$ times worse than optimal in the single-disk case. This bound applies directly to *reverse aggressive*, since there is no asymmetry between the reverse and forward problems in the single-disk case. It applies to the multiple-disk case as well, since a request sequence that contains only blocks that reside on a single disk is a special case.

3.2.4 Conservative*: lower bound*

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

Example: Suppose that F divides K, and also that d divides K, and consider a repeated cycle on $K + (\frac{K}{F} - 1)d$ blocks. *Conservative* always evicts the page just referenced whenever it fills a hole, since that is the page that will not be needed again for the longest time. Thus *conservative* will never be able to overlap prefetches with each other or with references. Since there are at least $(\frac{K}{F} - 1)d$ holes on each pass through the cycle, *conservative* will spend at least $K + (\frac{K}{F} - 1)d + (\frac{K}{F} - 1)dF$ steps on each pass through the cycle. Suppose that the blocks are colored such that each contiguous sequence of d blocks in the cycle contains one block from each of the d

disks. It is not hard to see that *opt* is able to maintain its holes in groups of d, one of each color, spaced F steps apart. Thus *opt* can service the entire sequence without stalling, and requires only $K + (\frac{K}{F} - 1)d$ steps on each pass through the cycle. The ratio of these two expressions (after a little manipulation) turns out to be at least as great as the stated bound.

3.2.5 Conservative*: upper bound*

Theorem 15 *On any reference string R, the elapsed time of* conservative *with d disks on R is at most $d+1$ times the elapsed time of the optimal prefetching strategy on R.*

Proof: Let m be the minimum number of fetches (which is exactly how many fetches *conservative* performs) on request sequence R. *Conservative*'s running time is at most $|R| + mF$, even if it never overlaps prefetches with each other or with the servicing of requests. Since the optimal algorithm *opt* must perform at least as many fetches as *conservative*, and also must service the request sequence R, *opt*'s running time is at least $\max(|R|, mF/d)$. The ratio of these is maximized with $|R| = mF/d$, and has the value $d+1$. □

3.2.6 Aggressive, fixed horizon, *and* forestall*: lower bound*

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

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

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

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

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

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

$$\cdots b_{d+1} \cdots b_{2d} b_{d+1} \cdots b_{2d-2} \cdots$$

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

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

Again, generalizing to arbitrary K/F is easy.

The bound applies to *fixed horizon* and *forestall* as well as to *aggressive*, since their respective conditions for initiating a prefetch are true at each time that *aggressive* initiates a prefetch in the above examples, and their prefetch and replacement decisions are the same as *aggressive*'s when their prefech conditions are true.

3.2.7 Aggressive: *upper bound*

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

Lemma 16 *If a set A of holes dominates a set B of holes, and some hole in A is filled and some hole at a larger index added to A, the resulting holes A' dominate B.*

Theorem 17 *On any reference string R, the elapsed time of* aggressive *with d disks on R is at most $d + \frac{(d+1)F}{K}$ times the elapsed time of the optimal prefetching strategy on R.*

Proof:

In the analysis of aggressive prefetching with one disk, it was shown that if A's holes dominate B's holes, A's cursor position is at least as great as B's, and each algorithm initiates a fetch, A's holes will continue to dominate B's when the fetch is completed. This result was referred to as the *domination lemma* [7]. The proof of this is similar to but simpler than that of Lemma 5 for algorithms working with the reverse sequence.

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

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

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

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

We show that for each i such that $0 \leq i < p-1$ where p is the number of phases (numbering the phases starting with 0), there are times T_i and T_i', such that

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

The theorem will follow from the first three conditions, as follows. For each phase i, *aggressive*'s elapsed time $e_{agg}(i)$ and *opt*'s elapsed time $e_{opt}(i)$ satisfy

$$e_{agg}(i) \leq de_{opt}(i) + (d+1)F$$

so that

$$\frac{e_{agg}(i)}{e_{opt}(i)} \leq d + \frac{(d+1)F}{e_{opt}(i)}.$$

Each phase except possibly the last is of length at least K, so that $e_{opt}(i) \geq K$. Putting these together, we have that for all phases but the last,

$$\frac{e_{agg}(i)}{e_{opt}(i)} \leq d + \frac{(d+1)F}{K}.$$

The last phase may be incomplete, i.e., may contain requests for fewer than K distinct blocks. *Aggressive* requires at most d times as many steps as *opt* to serve the last phase, as shown below.

This claim is proven by induction on i. The basis ($i = 0$) is trivial, since both algorithms start at the beginning of the first phase in the same state, with all disks idle.

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

We first show that for each index j in phase i, *aggressive*'s cursor passes j after at most d times as many steps as *opt*'s cursor takes to pass j. Let $T_i + t_A(j)$ be the time *aggressive* serves request j, and let $T_i' + t_O(j)$ be the time *opt* serves j. Assume by way of contradiction that *aggressive*'s cursor falls behind *opt*'s (relative to the start of the phase) by more than a factor of d, and let ℓ be the least index for which this happens, i.e., $t_A(\ell) > dt_O(\ell)$. It must be true that *aggressive* has a hole at ℓ (or equivalently stalls on the ℓ^{th} request in the phase) at time $T_i + t_A(\ell) - 1$, and that the ℓ^{th} request in the phase is in *opt*'s cache before time $T_i' + t_O(\ell)$, since $T_i + t_A(\ell)$ is the *first* time *aggressive*'s cursor falls behind *opt*'s by more than a factor of d. As noted previously, each disk of *aggressive*'s fills a hole every F steps as long as there are holes of that disk's color in the phase. Let h be the number of holes in $H^-_{agg}(T_i)$ that are the same color as the one at ℓ, up to and including the one at ℓ. Then $t_A(\ell) \leq hF$, since the hole at ℓ is filled at a time no later than $T_i + hF$. $H^+_{opt}(T_i')$ contains at least h holes at or before ℓ, since $H^-_{agg}(T_i)$ dominates $H^+_{opt}(T_i')$. Thus the earliest time *opt* could finish filling all its holes up to index ℓ is $T_i' + \lceil h/d \rceil F$, even if it fills a hole every F steps with each disk. Thus we have a contradiction: $hF \geq t_A(\ell) > dt_O(\ell) \geq d(\lceil h/d \rceil F) \geq hF$.

To show that *aggressive*'s holes after finishing phase i dominate *opt*'s holes, we need another induction. Let I_j' denote the F-step interval $[T_i' + jF, T_i' + (j+1)F)$,

$j \geq 0$, and let c_j be *opt*'s cursor position at time $T'_i + jF$, for each j such that *opt*'s cursor is still in phase i at time $T'_i + jF$. Let $I_j = [T_i + jdF, T_i + (j+1)dF)$. Consider the set of at most d fetches that *opt* initiates during I'_j. We match these to the set of fetches *aggressive* initiates during I_{j+1}. We prove by induction on j that $H^+_{opt}(T'_i + jF)$ is dominated by $H^+_{agg}(T_i + d(j+1)F)$. The base case follows from the hypothesis that $H^-_{agg}(T_i)$ dominates $H^+_{opt}(T'_i)$. Any fetches completed or initiated by *aggressive* during I_0 do not affect this, by Lemma 16. For the inductive step (on j), note that each fetch *opt* initiates during I'_j is initiated at a cursor position at most c_{j+1}, and that *aggressive*'s cursor position is at least c_{j+1} during the interval I_{j+1}. Thus *aggressive*'s fetches can be matched to *opt*'s and the domination lemma implies that *aggressive*'s resulting holes $H^+_{agg}(T_i + (j+2)dF)$ dominate *opt*'s resulting holes $H^+_{opt}(T'_i + (j+1)F)$. Any extra fetches of *aggressive* (there may actually be as many as d^2 by *aggressive* and as few as zero by *opt* during their respective time intervals) do not affect this, by Lemma 16. As a special case, if *aggressive* should stop fetching altogether at some time and thus have fewer than d fetches to match to *opt*'s, *aggressive* has reached the optimal cache configuration: its cache contains the next K distinct requests, and its holes are as far from the cursor as possible. These holes certainly dominate *opt*'s holes at any earlier cursor position.

Consider the value j^* such that *opt*'s cursor reaches phase $i+1$ during I'_{j^*}. Then by the preceding arguments, *aggressive*'s cursor reaches phase $i+1$ by time $T_i + (j^*+1)dF$ and *aggressive*'s holes $H^+_{agg}(T_i + (j^*+1)dF) = H^-_{agg}((j^*+1)dF + F)$ after completing all fetches initiated in I_{j^*} dominate *opt*'s holes $H^+_{opt}(T'_i + j^*F)$ after completing all fetches initiated in I'_{j^*-1}. Let $T_{i+1} = T_i + (j^*+1)dF + F$ and let $T'_{i+1} = T'_i + j^*F$, and the conditions for the induction step on the phase index i are met. □

3.3 The algorithms' running times

In this section we consider the time required to determine a prefetching schedule in the uniform-cost RAM model (see, for example, [3]). This is distinct from the time required to serve the sequence in the model described in Chapter 2, which is the primary measure we are trying to optimize.

First, consider the single-disk case. We assume that the i^{th} member of the set B of blocks is identified by the integer i. We will need per-block lists of requests (indices in the request sequence R); let $Next(b)$ refer to the head of the list of references to block b. Initially, $Next(b)$ points to the first request of block b; after that request is served, $Next(b)$ will be updated to point to the next occurrence of b in R, and so on. We will also need a vector *InCache* indexed by the set B indicating for each block whether or not it is present in the cache, and a pointer $NextHole$ indicating the index of the first hole in the request sequence. Finally, we will need a priority queue $Cache$ containing the identifiers of all blocks present in the cache, and keyed on the index in the request sequence of the next request to that block. $Cache$ will need to be augmented by an operation to update the key of an item (which could be implemented as a deletion and a reinsertion), as well as to the usual operations to insert items and delete the item with maximum key. Note $Cache$ will never contain more than K keys. Each operation on $Cache$ thus requires $O(\log K)$ time (see, for example, [3]). Note that the maximum element in $Cache$, the value of $NextHole$, and the position of the cursor provide the information needed by *aggressive*, *fixed horizon*, and *reverse aggressive* to decide when and what to prefetch, and what to evict.

A preprocessing step to initialize these data structures requires time linear in $|B| + |R|$; we assume $K \leq |B|$, since the scheduling problem is trivial otherwise. To maintain these structures when serving a request of block b, we need to update the pointer $Next(b)$ and update b's entry in the priority queue $Cache$. Thus scheduling the servicing of a request requires $O(\log K)$ time. To maintain these structures when evicting a block b_1 and fetching b_2, we delete the maximum element (which is b_1) from $Cache$, insert b_2 in $Cache$, update the vector *InCache* appropriately, and scan forward in R from $NextHole$ until a request is found that is missing from the cache (by referring to *InCache*); this index becomes the new $NextHole$. These operations require time $O(\log K)$ with the exception of the scan of the request sequence to find the new $NextHole$. The scans require $O(|R|)$ time, amortized over the entire sequence. $|R|$ is an upper bound on the total number of fetches. The reversal of R and of *reverse aggressive*'s reverse schedule can be done in time linear in $|R|$. Thus, each of the algorithms *aggressive*, *fixed horizon*, and *reverse aggressive* can be implemented to run in time $O(|B| + |R| \log K)$ in the uniform-cost RAM model.

A simple implementation of *conservative* is to run Belady's paging algorithm, recording each fetch/eviction pair along with a "release index," i.e., the index of the last request of the evicted block (before it is fetched back into the cache later in the schedule, if ever). A similar analysis to that above shows the same bound of $O(|B| + |R| \log K)$ for the construction of this list of fetches and evictions. The list can then be "played back" to construct a schedule for the fetches and the serving of the sequence, issuing each fetch as soon as the cursor has passed the release index and the disk is free. Thus, we have the same bound on *conservative*'s running time as on that of the other algorithms.

In the case of $d > 1$ disks, we assume a constant-time operation yields the disk a block resides on, given the block's identifier. The changes required in the analysis of *conservative* are trivial. For the other algorithms, data structures are maintained on a per-disk basis as needed. $NextHole$ becomes a vector of d entries for *aggressive* and *fixed horizon*. A linear time pre-processing step can be used to produce per-disk request sequences; these are needed to update $NextHole$. In the case of *reverse aggressive*, it is the priority queue $Cache$ that needs to be split into d separate structures, one for each disk; none will ever contain more than K keys. Thus, the running time bound given above applies to the multi-disk case as well as the single-disk case.

Finally, we consider the time required to evaluate *forestall*'s prefetch predicate $d_i \leq iF$. We use a set of priority queues $Holes$, one per disk, containing the index of the next request of each missing block that resides on the disk. The cursor position is a fixed value (at any given point in the schedule) which can be subtracted from the index of a hole to yield the distance to the hole. The priority queues of $Holes$ will need the usual priority queue operations $insert$ and $deletemin$, and a special operation $slack = \min_h(h - F \cdot rank(h))$, where h ranges over the set of keys stored (i.e., indices of holes). *Forestall*'s prefetch predicate is then $slack - cursor \leq 0$. Note that this data structure will never contain more than $|B|$ elements. *Forestall* also needs the data structures (and has the same running time components) as *aggressive* and *fixed horizon*. *Forestall*'s running time is thus $O(|B| + |R| \log K + T(|R|, |B|))$, where $T(n, m)$ is the time required to execute n operations on the data structure $Holes$ and m is the maximum number of elements. We leave open the problem of implementing this data structure in time $o(m)$ per operation. A trivial bound on $T(n, m)$ is $O(nm)$, yielding $O(|R||B|)$ for the running time of *forestall*.

Chapter 4

EXPERIMENTAL ANALYSIS

This chapter presents the results of joint work with Tomkins, Patterson, Bershad, Cao, Felten, Gibson, Karlin, and Li [24]. The presentation follows the chronological development of the results. An assessment was made of the practical algorithms *aggressive* and *fixed horizon*, using *reverse aggressive* as a benchmark against which to evaluate their performance. This comparison led to the search for a new algorithm with the best characteristics and none of the drawbacks of the others. *Forestall* is the result of that effort.

4.1 Overview of experimental results

In this chapter 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 *aggressive*, *fixed horizon*, *reverse aggressive*, and *forestall*. 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, full-lookahead case. 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 [8, 36].

4.2 Simulation model

Our theoretical model described and analyzed in Chapters 2 and 3 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 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. The following describes in detail these differences between the theoretical and simulation models, and several ways in which the algorithms are modified to account for them.

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 5.3). (The systematic effects of disk scheduling on disk response time are considered separately).

2. Access patterns exhibit locality of reference and data are striped across multiple disks in practice; the theoretical model allows worst-case data layouts and reference sequences.

   In practice, the combination of striped data layout and locality of reference balances loads across the disks. This allows *fixed horizon*, *aggressive*, and *forestall* to effectively utilize multiple disks and to 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 over *aggressive* since its late replacement decisions tend to lead to fewer 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*, *reverse aggressive*, and *forestall* 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*, *reverse aggressive*, and *forestall* to submit disk requests in batches. We have found that the performance of all the algorithms benefits from the CSCAN disk scheduling algorithm (see, for example, [40]).

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 holes 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 (which is identical to the fetch time F in the theoretical model). This ensures that any prefetch issued 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 of Patterson *et al.* [36]) yields a value of $H = 62$; we used this value in all our simulations, except where noted otherwise.

4.3 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. As mentioned, the prefetch horizon H is computed as the ratio of the average time it takes to read a block from disk to the minimum time it takes to consume a single block of data.

Aggressive: Whenever a disk D is free, construct a batch of at most `batchsize`

Table 4.1: Batch sizes used for *aggressive*.

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

fetches (see Table 4.1) 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 Chapter 2, 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 `batchsize` fetch/eviction pairs B_i/B'_i that have been released (or all that have been released, if there are fewer than `batchsize`), and for which B_i resides on disk D, to the batch. Issue the batch.[1]

---

[1] The batch sizes and estimate F used by *reverse aggressive* are discussed in Section 4.5.4.

We postpone the description of our implementation of *forestall* until Section 4.6, where it can be better motivated in light of the performance characteristics of the other algorithms.

4.4 Simulation environment

Trace-driven simulation was used 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 we use is unaffected by prefetching and caching activity, so that it makes sense to use the same trace with different prefetching and caching algorithms. 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, the simulators do not model serialization of memory bus transactions.[2]

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.* [26] (which is based on that of Ruemmler and Wilkes [38]) 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 4.2 lists several characteristics of the HP 97560 (taken from [38]). The CMU simulator uses the Berkeley RaidSim [9] 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 4.3 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

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

Table 4.2: HP 97560 characteristics.

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 4.3: Comparison of the simulators on the xds and synth traces.

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

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.

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.

4.4.1 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 in which 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 (cscopel) 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

Table 4.4: Trace summary data.

trace	reads	distinct blocks	compute time (sec)	avg. compute time per read (msec)
dinero	8867	986	103.5	11.7
cscope1	8673	1073	24.9	2.87
cscope2	20206	2462	37.1	1.84
cscope3	30200	3910	74.1	2.45
glimpse	27981	5247	38.7	1.38
ld	5881	2882	8.2	1.39
postgres-join	8896	3793	11.5	1.29
postgres-select	5044	3085	79.2	15.7
xds	10435	5392	30.8	2.95
synth	100000	2000	99.9	0.999

a non-indexed 3.2MB relation. The relations are those used in the Wisconsin Benchmark [16]. 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.

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 [16] 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 4.4 shows the length (number of read requests), number of distinct blocks requested, and 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.

4.4.2 *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 4.2. 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.

4.5 Performance of *aggressive*, *fixed horizon*, and *reverse aggressive*

In this section, we examine the behaviors of *aggressive*, *fixed horizon*, and *reverse aggressive* 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. The performance of *forestall* is reported in Section 4.6.

4.5.1 *Comparison with demand fetching*

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. Our implementation of demand fetching does not include the sequential readahead common to many file systems. However, the HP 97560 contains a readahead buffer, so that sequential accesses are served from the buffer (requiring only about 3 milliseconds per read) rather than from the disk itself. Figure 4.1 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. Each group of four bars represents the performance of

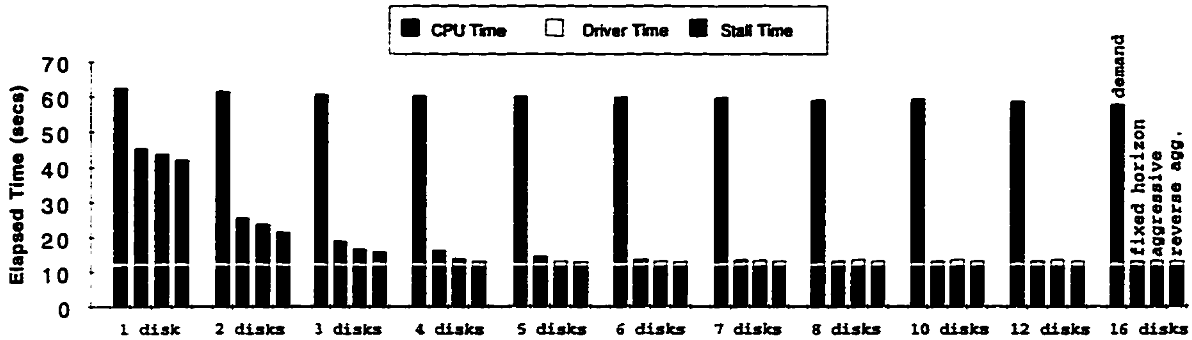


Figure 4.1: Performance on the postgres-select trace.

the four algorithms; they are, in left to right order, *optimal demand fetching*, *fixed horizon*, *aggressive* and *reverse aggressive*. 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.

4.5.2 Fundamental performance characterization of aggressive *and* fixed horizon

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 4.2 summarizes the results for one to four disks. Each group of three bars represents the performance of the three algorithms *fixed horizon*, *aggressive*, and *reverse aggressive*, in left-to-right order.

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

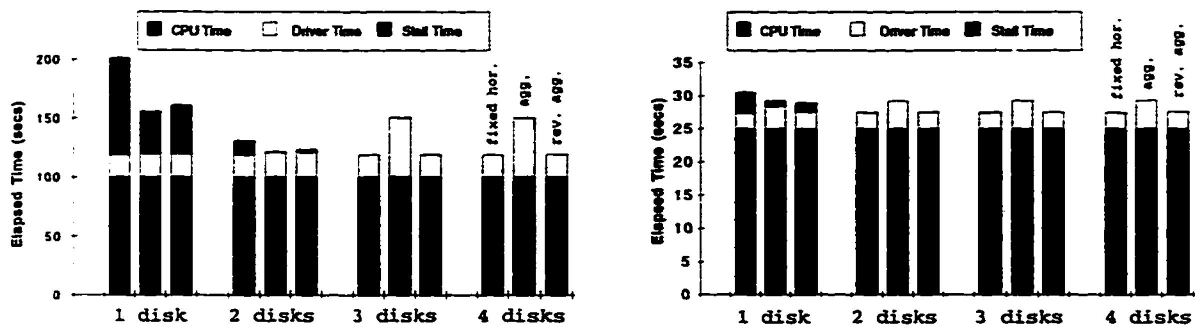


Figure 4.2: Performance on the synth (left) and cscope1 (right) traces.

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 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.

4.5.3 Performance of aggressive *and* fixed horizon *on 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 4.2 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 4.3 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.

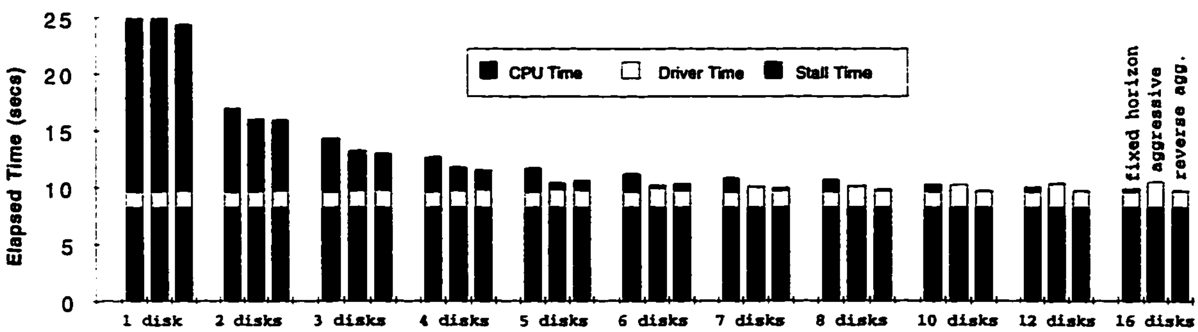


Figure 4.3: Performance on the ld trace.

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 4.4. Note that *reverse aggressive*'s performance is much

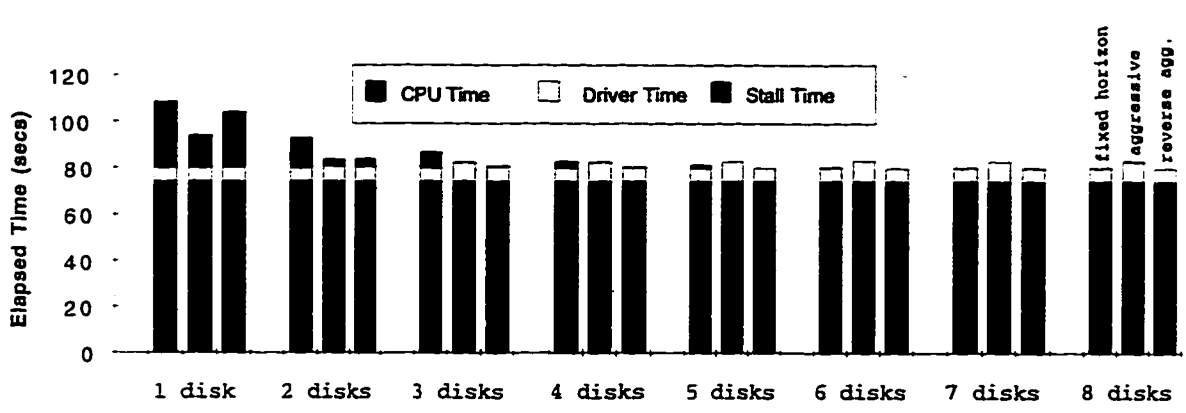


Figure 4.4: Performance on the cscope3 trace.

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 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

Table 4.5: Disk utilization on the postgres-select trace.

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

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 Appendix A.

Table 4.5 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.

4.5.4 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 brevity, we present general observations and only a small portion of the data. For the full data, see Appendix A.

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 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

Table 4.6: Percentage improvement of CSCAN over FCFS on the postgres-select 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

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).

Table 4.6 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 4.5 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.

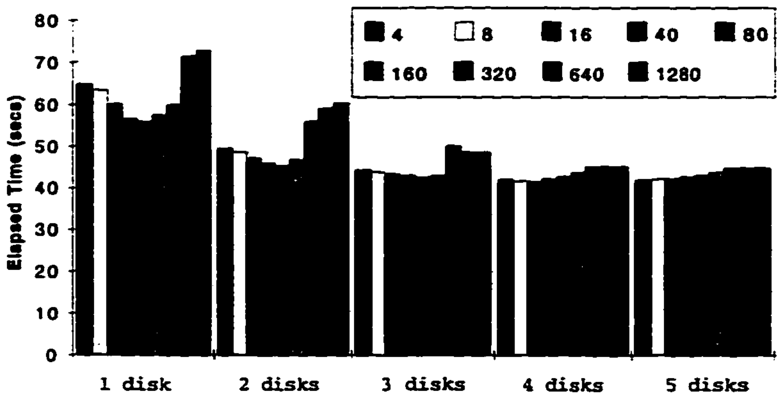


Figure 4.5: Performance of *aggressive* on the cscope2 trace, as a function of the batch size.

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 best 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 best batch size for one to three disks was 16, and for four or more was 4. All the results for *aggressive* presented in Section 4.5.3 were obtained using the batch sizes given in Table 4.1. 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 4.6 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

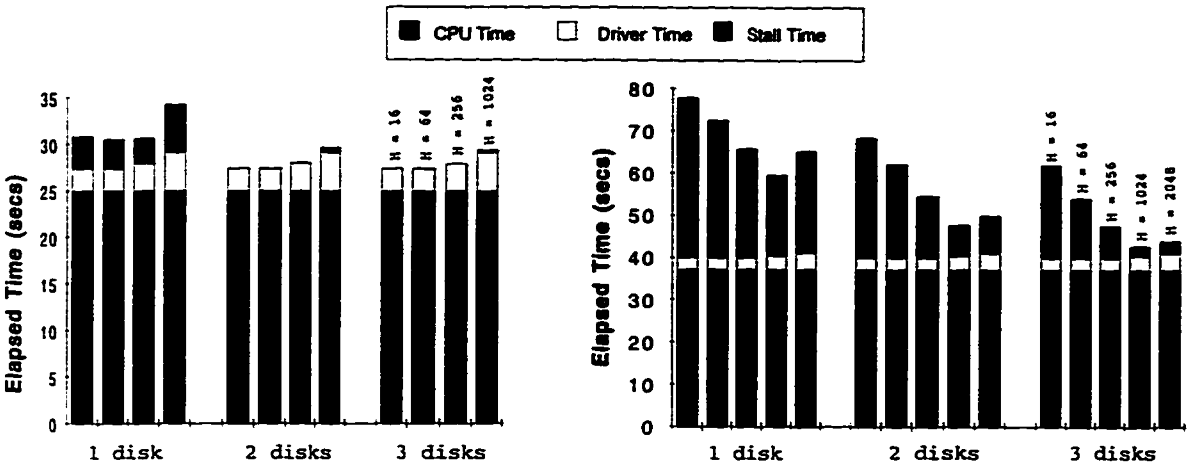


Figure 4.6: Performance of *fixed horizon* as a function of the prefetch horizon H on the cscope1 (left) and cscope2 (right) traces.

eliminated.

On the more I/O bound traces such as cscope2, also shown in Figure 4.6, 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 best configuration (choice of F and batch sizes) for each trace and each number of disks.

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

Table 4.7: 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.

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

Processor speed and cache size

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.

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 4.7, which shows the performance of *fixed horizon* relative to *aggressive* as a percentage difference, as a function of the cache size and the number of disks on the glimpse trace.

4.6 *Forestall*: an algorithm with the best features of the others

The simulation results of the previous section indicated the need for a new algorithm. It is desirable to have a single algorithm that has good performance regardless of I/O-boundedness or compute-boundedness, and that is simpler and more practical than *reverse aggressive*. The design of *forestall* to address these issues was described in Chapter 2.

4.6.1 *Implementation of* forestall

As do the other algorithms, *forestall* requires modifications to account for differences between the theoretical model and real systems. Requests need to be issued in batches 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 smaller value of $F' = F$ if the average disk access times is less than 5ms, and the larger value of $F' = 4F$ 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. A value of $F' =$ `batchsize` $\cdot F$ would eliminate this problem as well, so that the first hole is fetched in time even in the worst case of reordering by CSCAN. However, this would result in over-aggressive prefetching.

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 `batchsize` is similar to *aggressive*'s. We used for *forestall* the batch sizes given in Table 4.1.

4.6.2 Performance of forestall

Figure 4.7 shows the performance of the three practical algorithms, *fixed horizon*, *aggressive*, and *forestall*, on the synthetic trace and xds. Each group of bars represents

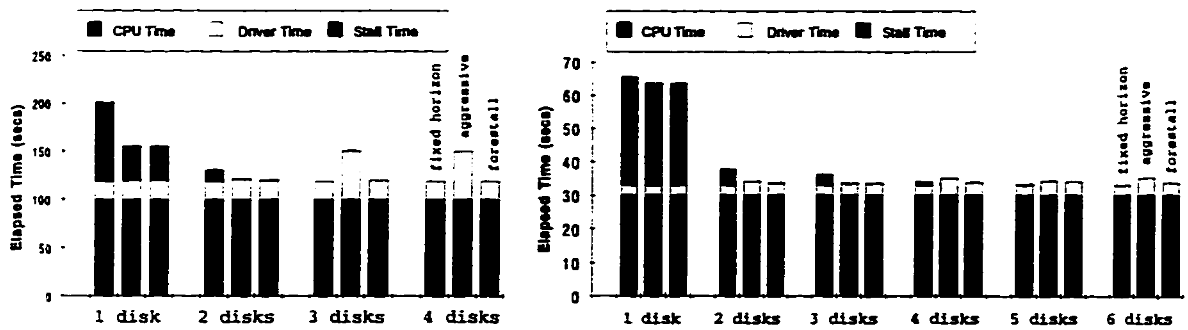


Figure 4.7: Performance on the synth (left) and xds (right) traces.

the performance of the three algorithms *fixed horizon*, *aggressive*, and *forestall*, in left-to-right order. *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 4.8 and 4.9 show the performance of the three algorithms on the cscope2 and glimpse traces. Once again, *forestall* has the best performance of the three

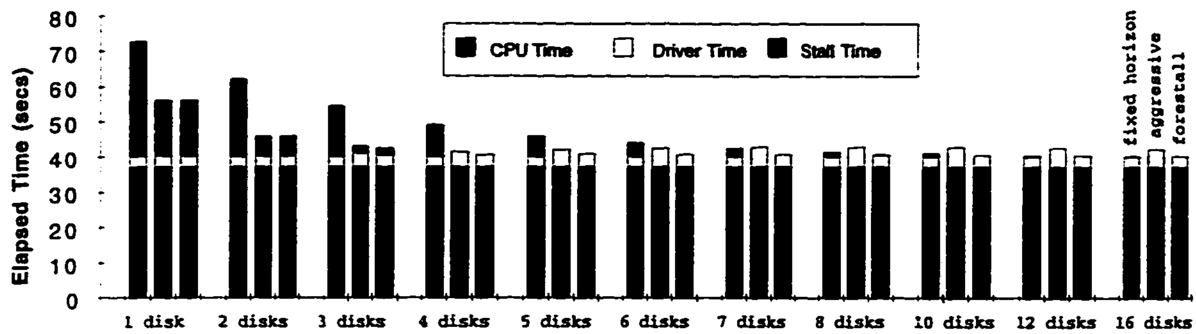


Figure 4.8: Performance on the cscope2 trace.

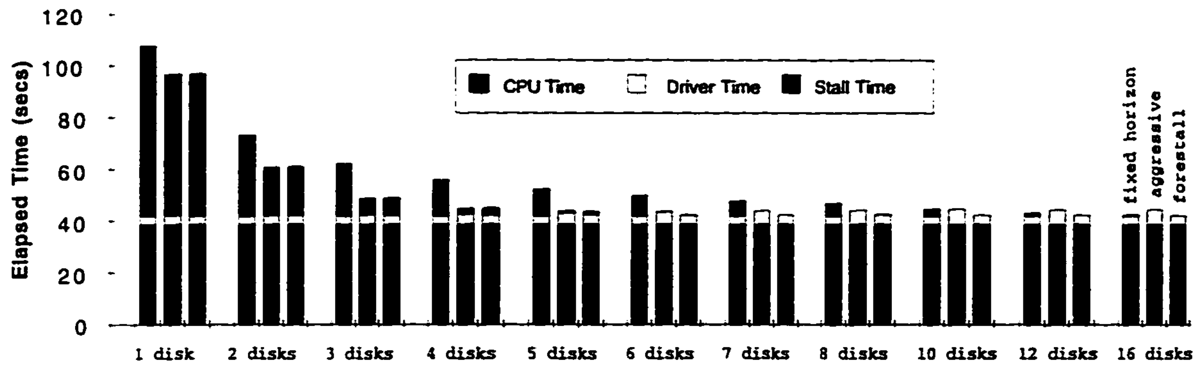


Figure 4.9: Performance on the glimpse trace.

practical algorithms. On all remaining traces, over all configurations, *forestall*'s performance was between 2% worse and 5.8% better than the better of *aggressive* and *fixed horizon* in that configuration. For the full data, see Appendix A.

Table 4.8 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*.

Table 4.8: Utilization of disks by *forestall* on the postgres-select trace.

disks	1	2	3	4	5	6
util.	.99	.92	.87	.81	.68	.63
disks	7	8	10	12	16	
util.	.62	.54	.39	.30	.22	

Chapter 5

AN EXACT SOLUTION TO A RESTRICTED PROBLEM

Cao *et al.* [7] studied the problem of integrated prefetching and caching with a single backing store. They left open the problem of finding an optimal schedule in time polynomial in both the input length and the cache size. We make partial progress on this problem in this chapter. We present an algorithm that finds a schedule with zero stall time if one exists.

We allow an arbitrary subset of the set of blocks B (specified as part of the input) to be present in the cache initially. It is necessary that some subset of B is initially present in the cache in order to avoid stalling completely; otherwise, the first request would stall. However, it is not hard to show that if the cache is empty initially, an optimal schedule can be obtained by fetching the first K distinct blocks in the request sequence R during the first FK units of time. The algorithm given here will then determine whether the remainder of the sequence can be served without any additional stall time.

5.1 Identifying optimal intermediate states

Recall the four properties that can be assumed of any optimal strategy in the single-disk case:

Optimal fetching: when fetching, always fetch the missing block that will be referenced soonest;

Optimal eviction: 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;

First opportunity: never evict A to fetch B when the same thing could have been done one time unit earlier.

Any schedule that does not follow these rules can be transformed into one that does, with performance at least as good. The first two rules specify what to fetch and what to evict, once a decision to fetch has been made. The last two rules constrain the times at which a fetch can be initiated. Unfortunately, these rules do not specify how to choose between an earlier prefetch with a correspondingly earlier eviction and a later prefetch with a correspondingly later eviction. The former helps prevent stalling on earlier holes, whereas the latter may help avoid the introduction of holes due to superior eviction choices, and hence prevent stalling at a later time.

In the following, we describe an algorithm that determines whether a sequence can be served with no stall time, given an initial cache state (i.e., a set of at most K blocks initially contained in the cache). Notice that since we are looking for a schedule to serve the sequence with no stall time, as we construct a schedule we may assume that the time index and the cursor position are the same. Thus the state of an intermediate point in a schedule is completely determined by the cursor position, the set of blocks contained in the cache, the block currently being fetched (if any), and the number of steps (from 0 to $F-1$) remaining until the fetch completes. Let $n = |R|$. We denote by $C_{i,f}$ the set of states that can be reached after i steps without stalling such that there are f steps remaining until the current fetch completes, where $0 \leq f \leq F-1$ and $0 \leq i \leq n$. $C_{0,0}$ contains exactly one state: the cache contains whichever set of blocks is specified by the input, and there is no fetch in progress initially. $C_{0,f}$ is empty for $1 \leq f \leq F-1$. The sequence can be served without stalling if and only if $C_{n,0} \neq \emptyset$.

Recall the notion of *domination*.

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

Definition: If the set A of holes in state c_A dominates the set B of holes in state c_B, and the cursor position of state c_A is at least as great as that of c_B, and the number of steps remaining on the current fetch of c_A (if any) is no greater than that of c_B, then we will say that c_A is *at least as good as* c_B.

Notice that, like domination, this relation is transitive. It is not hard to show that given two states c_A and c_B, if c_A is at least as good as c_B and there is a schedule that completes the sequence in time t starting from state c_B, then there is a schedule that completes the sequence in time at most t starting from state c_A.

Theorem 18 *For every (i, f), where $0 \leq f \leq F-1$ and $0 \leq i \leq n$, either $C_{i,f}$ is empty, or contains a nonempty subset $C^*_{i,f}$ such that any state $c^*_{i,f} \in C^*_{i,f}$ is at least as good as any state $c_{i,f} \in C_{i,f}$.*

For every $c_{i,f}$, if there is a schedule that passes through $c_{i,f}$ and does not stall, then there is a schedule that passes through any $c^*_{i,f} \in C^*_{i,f}$ and does not stall. Thus we can restrict our attention to finding in polynomial time a unique representative $c^*_{i,f} \in C^*_{i,f}$ for each pair (i, f), if one exists. The following proof of theorem 18 outlines such an algorithm.

Proof: We prove the existence of a unique polynomial time computable $c^*_{i,f}$ by induction on i. The basis, $i = 0$, is trivial. For the induction $(0 < i \leq n)$, we consider three cases.

First, consider the case $0 < f < F-1$. By the induction hypothesis, either $C_{i-1,f+1} = \emptyset$ or we can find some $c^*_{i-1,f+1}$. If the former, then $C_{i,f} = \emptyset$; i.e., if there is no nonstalling schedule that reaches cursor position $i-1$ with $f+1$ steps remaining on a fetch, then there is no nonstalling schedule that reaches cursor position i with f steps remaining on a fetch. If the latter, then $C_{i,f}$ may still be empty; this is the case if $c^*_{i-1,f+1}$ leads to a stall at step i because r_{i-1} is missing from the cache. Otherwise, we can take $c^*_{i,f}$ to be the same state as $c^*_{i-1,f+1}$ with the cursor advanced by one position and one fewer step remaining until the fetch completes, since every schedule that passes through $c^*_{i-1,f+1}$ must pass through this state.

Next, consider the case $f = F-1$. If $C_{i-1,0} = \emptyset$, then $C_{i,F-1} = \emptyset$. If $C_{i-1,0} \neq \emptyset$, then $C_{i,F-1}$ may be nonempty. As in the previous case, $c^*_{i-1,0}$ may lead to a stall at step i. Otherwise, by the optimal prefetching rules described previously, $c^*_{i,F-1}$ can be taken to be the state that results from evicting the block not needed for the longest time among all blocks in the cache when the state is $c^*_{i-1,0}$ and initiating a prefetch for the first missing block.

Finally, consider the case $f = 0$. There are two ways to reach a state in $C_{i,0}$. The previous state may be in $C_{i-1,0}$ or $C_{i-1,1}$. If $C_{i-1,0}$ and $C_{i-1,1}$ are both empty, then so is $C_{i,0}$. If either $c^*_{i-1,0}$ or $c^*_{i-1,1}$ exists but leads to a stall at step i and the other does not exist, or both exist but lead to a stall at step i, again $C_{i,0}$ is empty. If only one exists and does not lead to a stall at step i, $c^*_{i,0}$ is easily identified similarly to the previous cases. Finally, if both $C_{i-1,0}$ and $C_{i-1,1}$ are nonempty and neither $c^*_{i-1,0}$ nor $c^*_{i-1,1}$ leads to a stall at step i, we may take $c^*_{i,0}$ to be the successor of $c^*_{i-1,1}$. To see this, consider the two sequences of states

$$c^*_{i-F-1,0} \rightarrow c^*_{i-F,F-1} \rightarrow \cdots \rightarrow c^*_{i-1,0} \rightarrow c_{i,0}$$

and

$$c^*_{i-F-1,0} \rightarrow c'_{i-F,0} \rightarrow c'_{i-F+1,F-1} \cdots \rightarrow c'_{i-1,1} \rightarrow c'_{i,0}.$$

Notice that at time $i-F$, the set of eviction choices available in state $c'_{i-F,0}$ includes all those available at time $i-F-1$ in state $c^*_{i-F-1,0}$, plus one more, possibly better choice: the block referenced at time $i-F-1$. The set of holes resulting from the eviction at time $i-F$ dominates that resulting from the eviction at time $i-F-1$, by the Domination Lemma of [7]. Thus $c'_{i,0}$ is at least as good as $c_{i,0}$. By choosing the successor of $c^*_{i-1,1}$ for $c^*_{i,0}$, we obtain a state at least as good as $c'_{i,0}$, which is at least as good as $c_{i,0}$. ($c^*_{i,0}$ may actually be different from and possibly better than $c'_{i,0}$, since it may be reached by a path through $c^*_{i-F-1,1}$ rather than $c^*_{i-F-1,0}$. The alert reader may notice that this choice may lead to a schedule which violates the rule *first opportunity*. The schedule can be transformed easily into one that does not violate the rule.) □

5.2 The algorithm

The preceding inductive proof implicitly defines a directed graph G on the set of vertices $\{(i,f)|0 \leq f \leq F-1, 0 \leq i \leq n\}$. Each node represents a state $c^*_{i,f}$ that is reachable without stalling. Each edge represents a single state transition (i.e., the changes that occur during one time unit) in a set of schedules that do not stall (one for each path that starts at $(0,0)$ and passes through the edge). If the state represented by a node (i,f) leads to a stall at step $i+1$, there are no edges out of the node.

The graph and the state associated with each node are computed easily from the request sequence and the initial cache contents, using the optimal prefetching rules to determine cache content changes. A zero-stall schedule exists if and only if there is a path from $(0,0)$ to $(n,0)$. A zero-stall schedule is easily computed from such a path.

The algorithm can be implemented in time $O(|B| + |R|\log K + F|R|)$ as follows. We consider the nodes of the graph G to be arranged in $|R| + 1$ columns, each corresponding to a time index (or cursor position) and containing F nodes. Each position in a column corresponds to a number between 0 and $F-1$ of steps remaining until a fetch completes. G can be constructed one column at a time in a single forward pass over the request sequence. Only F different cache states need be maintained. Moreover, in constructing column $i+1$ from column i, only one of the F states requires a nontrivial change: the one which results from initiating a fetch at time i. The data structures required to maintain the state information were described in Section 3.3. A similar analysis to that of Section 3.3 yields the stated bound. The term $|R|\log K$ comes from the (at most) one nontrivial cache state change per column. The term $F|R|$ reflects the fact that F different *NextHole* pointers need to be maintained. Searching for a path (and schedule) once G is constructed requires time $O(F|R|)$ as well, for instance, by depth-first search.

The algorithm is easily extended by a brute force approach to determine whether a sequence can be served with stall time c or less for any constant c, with a multiplicative increase in running time of $O(|R|^c)$. This is done by inserting c dummy requests into the request sequence, one for each unit of stall time. There are $O(n^c)$ ways to insert c dummy requests in a sequence of length n.

Chapter 6

INTEGRATING PREFETCHING WITH PROCESSOR AND DISK SCHEDULING

Integrated prefetching and caching policies, augmented by mechanisms to allocate cache space among multiple processes, have been shown empirically to improve the performance of multi-programmed workloads [8, 36, 43]. These studies used standard multi-programming scheduling mechanisms to arbitrate the processes' competing prefetch requests and processing demands. The goal was to improve average I/O response time, leaving the scheduling policies unchanged.

A natural question arises: can performance improve further if we assume that not only are prefetching and caching decisions under the control of a single manager, but that the scheduling of I/O resources (i.e., the order in which different processes' prefetch requests are served) and processing resources (the order in which the processes' computations are executed) are integrated into the policy as well? In this chapter, we consider the problem of minimizing the total elapsed time of a set of independent I/O-intensive processes. In scheduling terminology, this is referred to as minimizing the makespan, or maximizing system throughput.

Consider the following example. Suppose each of two processes, P and Q, start at time 0 and request data from a single disk in a cyclic fashion. For simplicity, suppose each process uses a data file that consists of only two blocks. Thus, process P issues the request sequence $p_1, p_2, p_1, p_2, \ldots$ and Q issues the request sequence $q_1, q_2, q_1, q_2, \ldots$. Suppose further that F units of time are required to fetch a block of data into the cache, and that it takes 1 unit of time to serve each request (i.e., each process computes for 1 unit of time after each request for a block of data). The requested block must be present in the cache for the computation to proceed, of course.

Suppose P's and Q's data blocks are brought into the cache alternately, say in the order p_1, q_1, p_2, q_2; these fetches complete at times F, $2F$, $3F$, and $4F$, respectively.

Until P's last block of data is available at time $3F$, only 2 units of work can be completed by the CPU. P is able to complete one unit of work (on block p_1 at time $F+1$) and Q completes one unit of work (on block q_1 at time $2F+1$). Finally at time $3F$, P's entire data set is resident in the cache and it can run unhindered by I/O stalls. The same happens for Q at time $4F$.

If instead we favor one of the processes, say P, by devoting resources to it exclusively, we reach a state sooner in which the processor can be fully utilized. If the blocks are fetched in the order p_1, p_2, q_1, q_2, P can run without stalling on I/O starting at time $2F$. Q's full data set still becomes resident at time $4F$.

A simplified version of this integrated scheduling problem reduces to the following combinatorial problem. (The reduction is outlined in Section 6.3; full details are given in Section 6.7.) Imagine that you are given a set of several independent streams of transactions (drafts and deposits) on a checking account that is backed by an unlimited savings account. Any time the checking account is overdrawn, the overdraft must be covered from savings. Your goal is to produce an ordering of the transactions that

1. respects the orders of the individual streams, and
2. minimizes the amount that has to be transferred from savings to cover overdrafts in the checking account.

In Section 6.6, we present a simple and efficient algorithm that finds an exact solution. The algorithm requires $O(n \log m)$ arithmetic operations, where n is the total number of transactions in all of the m sequences. An algorithm that solves this problem and has the same running time was given by Abdel-Wahab and Kameda [1]. However, as described in Section 6.4, the algorithm given here is somewhat better suited to the integrated scheduling problem.

6.1 Motivation and background

Suppose multiple processes share a cache backed by a storage device, and that the system has advance knowledge of the processes' sequences of requests for items residing on the backing store. Suppose there are no constraints regarding the interleaved servicing of the requests of the different processes, other than that the requests of an individual process must be served in order. The goal is to minimize the completion time of the last process to finish.

As we have seen, the problem described above appears to be a difficult one to solve exactly, even in the case of a single process and a single storage device. To make the multiple-process problem tractable, we restrict our attention to the single-disk case. We further simplify the problem by ignoring the question of eviction choices. Although this seems unrealistic, the *forestall* algorithm described in Section 2.4.5 determines whether to prefetch based only on its estimate of the likelihood of a stall given its current cache state. The algorithm does not weigh this likelihood against the benefits of waiting to prefetch to make a better eviction decision, and thus possibly avoid stalling at a later point in the schedule. We have seen in the experiments of Chapter 4 that this algorithm performs at least as well as an algorithm that is provably near-optimal in a simplified model of a prefetching and caching file system. This suggests that in practice, it is not necessary to solve the difficult problem of determining exactly when each prefetch should occur based on an optimal or near-optimal sequence of eviction choices. It is sufficient to prefetch whenever a stall is imminent given the current cache contents, and to delay prefetching if the cache contains the blocks needed in the near future so that no stall is imminent.

Thus, in the more difficult multiple process case, the algorithm given here may well lead to a practical prefetch scheduler when combined with the *forestall* algorithm. This is despite the fact that it is designed without considering the effects of cache evictions, as will become clear in Section 6.2. A simple modification of *forestall*, which takes into account all m processes' request sequences and the holes in them, can determine on a global basis whether prefetching is needed to keep the processor busy. Recall *forestall*'s inequality $d_i \leq iF$ which determines when prefetching is needed to avoid a stall, where d_i denotes the distance from the cursor to the i^{th} hole in the (single) request sequence, and F is the fetch time. Prefetching is needed if this inequality is true for any i and the *do no harm* rule allows it.

For m request sequences, prefetching is needed if

$$\sum_{j=1}^{m} \min_i (d_{j,i} - iF) \leq 0$$

where $d_{j,i}$ denotes the distance from the j^{th} sequence's cursor to the i^{th} hole in the j^{th} sequence; otherwise, there are enough blocks in the cache that are needed in the near future for at least one cursor to continue advancing. The algorithm given in this chapter can determine an order in which to prefetch the multiple processes' blocks, once the decision to prefetch has been made.

Notice that we have not completely specified an algorithm for the integration of all the resource scheduling problems considered. In particular, we have not specified a mechanism for choosing which cursor to advance (i.e., which process to run) in the event that more than one process has its next request available in the cache. In the special case in which no cache evictions are necessary, an arbitrary choice can be made, and the solution produced by the algorithm of this chapter is still optimal. In the general case, this choice affects the possibilities for evictions. Moreover, evictions change the set of holes, which is the input to our algorithm that determines the order in which to fill them. We have two problems that interact in a complex way. Given an interleaving of the multiple request sequences into a single sequence, we can use *forestall* or one of the other algorithms analyzed in previous chapters to determine a good schedule for prefetching and caching. Such a schedule fixes a complete list of holes (partially determined by its eviction choices) that must be filled over the full lifetimes of all the request sequences. Given such a complete list of holes, the algorithm of this chapter determines an ordering in which to fill them and a partial ordering of the requests; any interleaving of the requests that is consistent with the partial ordering is optimal. Which of these problems is the chicken and which is the egg?

One possible solution to this dilemma is to alternately run a prefetching and caching algorithm and the interleaving algorithm of this chapter. Interesting open questions are whether this process will converge, and if so, the quality of the schedule produced.

Another, more practical, possibility is to use *forestall* as described above, along with a mechanism such as the *LRU-SP* policy of Cao *et al.* or the *cost-benefit* policy of Patterson *et al.* to determine a victim process to give up a block for eviction each time a block is prefetched; the *optimal eviction* rule will determine which of that process' blocks to replace. The interleaving algorithm proposed in this chapter could be used to schedule prefetch operations (and partially constrain cursor movements) incrementally in batches of requests for K distinct blocks (in all sequences together). We have noted in Chapter 3 that it is never necessary to create a new hole among the next K distinct blocks to be served, so that for batches of this size, the assumption of no cache evictions is valid. We will also need a mechanism for determining how many of the K blocks in a batch are devoted to each process.

6.2 Formal problem statement

The general problem (including cache evictions) is formalized as follows:

- Let $B = B^1 \cup \ldots \cup B^m$ be a collection of disjoint sets of blocks residing on the backing store.
- A *reference sequence*, or *request sequence*, is an ordered sequence of references $R^k = r_1^k, r_2^k, \ldots r_{|R^k|}^k$, where each $r_i^k \in B^k$.
- There are m separate reference sequences $R^1, \ldots, R^m$.
- There is a cache of size K that contains at most K blocks in B at any time.
- Fetching a block from a disk into the cache takes F time units.

The references in each sequence R^k must be *served* in order. A single reference can be served in one unit of time. However, for a reference to be served, it must be in the cache. We imagine that for each reference sequence there is a *cursor* that at any time points to the next request to be served. If this request is for a block that is in the cache, the cursor can be advanced by one request during the next time unit. If several cursors point to blocks that are present in the cache, one and only one of them can be advanced in a single time unit. If all requests pointed to by the cursors are for blocks that are not in the cache, processing *stalls* until one of the missing blocks arrives in the cache (i.e., until the fetch for that block completes). Note that, to the extent that the cursors are advancing, prefetches can overlap the serving of requests.

There are two constraints on the prefetches performed:

1. If a fetch of block b is initiated at time t and the cache contains K blocks at that time, some block b' in the cache must be *evicted* to make room for the incoming block. Neither the fetched block b nor the evicted block b' is available during the F time units between t and $t + F$ in which the fetch occurs.

2. The fetches are sequential: If a fetch is initiated for a block at time t, no other fetch can be initiated until time $t' \geq t + F$.

The goal of a multi-process prefetching and caching algorithm is to construct, on input request sequences $\{R^k\}$, a schedule for prefetching and serving requests that minimizes the elapsed time required to serve all of the R^k; this elapsed time is equal to $\sum_{k=1}^{m} |R^k|$ plus the total stall time.

The schedule specifies

- which blocks to fetch,
- when to fetch them,
- which cache blocks to evict, and
- when to service each request.

We solve this problem for the special case of an unbounded cache; that is, we assume cache evictions are never necessary. However, for reasons described in the previous section, we believe that this algorithm is nonetheless practical. We will thus be concerned only with the order in which to fetch blocks into the cache, and not which blocks to evict. The algorithm presented here will work no matter what set of blocks is contained in the cache initially.

At any time during the processing of the requests, for each request sequence there is some distance from the cursor to the first hole in that sequence. We refer to the requests at or following the cursor and preceding the first hole as *uncovered*. Uncovered requests can be thought of as work available to the processor; if a total of U requests are uncovered in all sequences, then the cursors can be advanced a total of U times before all cursors reach holes and a stall may be incurred. Clearly, an incrementally in batches of requests for K distinct blocks (in all sequences together). We have noted in Chapter 3 that it is never necessary to create a new hole among the next K distinct blocks to be served, so that for batches of this size, the assumption of no cache evictions is valid. We will also need a mechanism for determining how many of the K blocks in a batch are devoted to each process.

6.2 Formal problem statement

The general problem (including cache evictions) is formalized as follows:

- Let $B = B^1 \cup \ldots \cup B^m$ be a collection of disjoint sets of blocks residing on the backing store.
- A *reference sequence*, or *request sequence*, is an ordered sequence of references $R^k = r^k_1, r^k_2, \ldots r^k_{|R^k|}$, where each $r^k_i \in B^k$.
- There are m separate reference sequences $R^1, \ldots, R^m$.
- There is a cache of size K that contains at most K blocks in B at any time.
- Fetching a block from a disk into the cache takes F time units.

The references in each sequence R^k must be *served* in order. A single reference can be served in one unit of time. However, for a reference to be served, it must be in the cache. We imagine that for each reference sequence there is a *cursor* that at any time points to the next request to be served. If this request is for a block that is in the cache, the cursor can be advanced by one request during the next time unit. If several cursors point to blocks that are present in the cache, one and only one of them can be advanced in a single time unit. If all requests pointed to by the cursors are for blocks that are not in the cache, processing *stalls* until one of the missing blocks arrives in the cache (i.e., until the fetch for that block completes). Note that, to the extent that the cursors are advancing, prefetches can overlap the serving of requests.

There are two constraints on the prefetches performed:

has to be transferred from savings to cover overdrafts in the checking account. (Each time a dollar is moved from savings into checking to cover a check, one unit of stall time is incurred.) We will give a more formal description of this reduction in Section 6.7; first, we introduce some notation and derive a solution to the new problem.

6.4 Reduced problem statement

Definition: Given a sequence $w = w_1 \ldots w_n$ of real numbers, the *depth* $D(w)$ of w is

$$\min_{0 \leq i \leq n} \sum_{j=1}^{i} w_j$$

and the *net* $N(w)$ of w is

$$\sum_{i=1}^{n} w_i.$$

We denote the net $N(w_1 \ldots w_i)$ of a prefix $w_1 \ldots w_i$ of w by $N(w, i)$ and similarly denote the depth of a prefix; we will also speak of the "depth of w at index i" or the "net of w at index i" when the meaning is clear. Notice that in comparing two sequences, the sequence whose depth is a greater number is the *shallower* of the two, since a sequence's depth is a non-positive number.

Given a set $W = \{w^k = w_1^k \ldots w_{n_k}^k : 1 \leq k \leq m\}$ of m sequences of numbers, an *interleaving* I of W is a sequence $I_1 \ldots I_n$, where $n = \sum_{k=1}^{m} n_k$, such that there is a one-to-one map M from

$$\{(k, i) : 1 \leq k \leq m, 1 \leq i \leq n_k\}$$

to $[1..n]$ such that

1. for all $1 \leq k \leq m$, for all $1 \leq i_1 < i_2 \leq n_k$, $M(k, i_1) < M(k, i_2)$, and

2. for all $1 \leq k \leq m$, for all $1 \leq i \leq n_k$, $I_{M(k,i)} = w_i^k$.

For a sequence $w = w_1 \ldots w_n$, let $B(w)$ and $R(w)$ denote the shortest non-empty prefix (if it exists) of w that sums to a non-negative value and the remaining suffix, respectively; that is, $B(w) = w_1 \ldots w_l$ and $R(w) = w_{l+1} \ldots w_n$, where

$$l = \min_{1 \leq i \leq n} \{i : N(w, i) \geq 0\}$$

optimal prefetching algorithm can be assumed to always fill the first hole (fetch the first missing block) in some request stream. Since we assume the cache is infinite, it never pays to wait and leave the storage device idle before initiating a prefetch; this would help only to make a better eviction decision if the cache were bounded. Thus we assume that a fetch is initiated at time 0, and every F time units thereafter, until the last hole is filled.

6.3 A reduction

As mentioned previously, to minimize the overall completion time, we can focus on minimizing the total stall time, i.e., the number of time steps during which no request is served (because all cursors are blocked by holes and no request can be served until the current fetch completes). If filling a hole in some sequence uncovers F requests (including the hole and all requests up to but not including the next hole in the sequence), the schedule "breaks even" in terms of uncovered requests, since it takes F steps to fill the hole, and F uncovered requests for blocks already present in the cache can be served concurrently. Any greater number represents a net gain of uncovered requests from the time at which the fetch is initiated until it completes; fewer than F requests uncovered will decrease the amount of work available to the processor, and increase the chance of a stall. If there are $U < F$ uncovered requests at some time iF (i.e. the i^{th} fetch has just completed), then only the U uncovered requests can be served before the next fetch completes at time $(i+1)F$ and more requests are uncovered; $F - U$ steps will be spent stalling.

We thus consider each request sequence to be simply a sequence of numbers. Corresponding to a request sequence containing u_1 uncovered requests, followed by a hole, then $u_2 - 1$ cached blocks and then another hole, etc., we have the sequence $u_1, -F, u_2, -F, \ldots$. Intuitively, it costs F steps to fill a hole; this cost must be paid before the benefit (that of being able to serve the requests uncovered) can be reaped. The problem reduces to the following problem: We are given a set of several independent streams of transactions (drafts and deposits) on a checking account, which is backed by a savings account. Whenever the checking account is overdrawn, the deficit must be made up out of savings. We need to interleave the transactions in an order respecting the orders of the individual streams and minimizing the amount that has to be transferred from savings to cover overdrafts in the checking account. (Each time a dollar is moved from savings into checking to cover a check, one unit of stall time is incurred.) We will give a more formal description of this reduction in Section 6.7; first, we introduce some notation and derive a solution to the new problem.

6.4 Reduced problem statement

Definition: Given a sequence $w = w_1 \dots w_n$ of real numbers, the *depth* $D(w)$ of w is

$$\min_{0 \leq i \leq n} \sum_{j=1}^{i} w_j$$

and the *net* $N(w)$ of w is

$$\sum_{i=1}^{n} w_i.$$

We denote the net $N(w_1 \dots w_i)$ of a prefix $w_1 \dots w_i$ of w by $N(w, i)$ and similarly denote the depth of a prefix; we will also speak of the "depth of w at index i" or the "net of w at index i" when the meaning is clear. Notice that in comparing two sequences, the sequence whose depth is a greater number is the *shallower* of the two, since a sequence's depth is a non-positive number.

Given a set $W = \{w^k = w_1^k \dots w_{n_k}^k : 1 \leq k \leq m\}$ of m sequences of numbers, an *interleaving* I of W is a sequence $I_1 \dots I_n$, where $n = \sum_{k=1}^{m} n_k$, such that there is a one-to-one map M from

$$\{(k, i) : 1 \leq k \leq m, 1 \leq i \leq n_k\}$$

to $[1..n]$ such that

1. for all $1 \leq k \leq m$, for all $1 \leq i_1 < i_2 \leq n_k$, $M(k, i_1) < M(k, i_2)$, and
2. for all $1 \leq k \leq m$, for all $1 \leq i \leq n_k$, $I_{M(k,i)} = w_i^k$.

For a sequence $w = w_1 \dots w_n$, let $B(w)$ and $R(w)$ denote the shortest non-empty prefix (if it exists) of w that sums to a non-negative value and the remaining suffix, respectively; that is, $B(w) = w_1 \dots w_l$ and $R(w) = w_{l+1} \dots w_n$, where

$$l = \min_{1 \leq i \leq n} \{i : N(w, i) \geq 0\}$$

if $\{i : N(w_1 \ldots w_i) \geq 0 \ \& \ \ 1 \leq i \leq n\} \neq \emptyset$.

Lemma 19 *Any suffix $w_i \ldots w_{|B(w)|}$ of $B(w)$ sums to a non-negative value.*

Proof: Since $B(w)$ is the shortest non-empty prefix of w with a non-negative net, $N(w, i-1) \leq 0$, with equality holding only in the case $i = 1$. (For any i, we take $w_i \ldots w_{i-1}$ to be the empty sequence.) □

Definition: Let $\mathcal{I}(W)$ denote the set of all interleavings of W. We seek an interleaving I such that $D(I)$ is maximum. Let $\mathcal{I}^*(W)$ denote the set of optimal interleavings; that is,

$$\mathcal{I}^*(W) = \{I \in \mathcal{I}(W) : D(I) = D^*(W)\}$$

where

$$D^*(W) = \max_{I \in \mathcal{I}(W)} D(I).$$

An algorithm that solves this problem and is very similar to the one described here, with the same running time, was given by Abdel-Wahab and Kameda [1]. However, their algorithm is fully offline; that is, it considers its entire input before producing any of its output. The algorithm given here constructs its solution incrementally, at least until it reaches a point at which it can not avoid "going into the hole," i.e., scheduling prefetches that uncover fewer than F requests. This makes the algorithm more suited to the multiple-process prefetching and caching problem. While the processor is not stalling (i.e., there are requests uncovered), it is desirable to avoid scheduling overhead. Once the processor begins to stall, scheduling overhead is less costly, or even free if it is entirely overlapped with I/O (which is likely).

6.5 Solving the reduced problem

The algorithm to find a "shallowest" interleaving is the following: consider each of the input sequences, and choose that one (call it w) with the shallowest prefix $B(w)$, i.e., choose w so that $|D(B(w))|$ is minimum. Output $B(w)$, replace w by the suffix $R(w)$ that remains after removing $B(w)$, and repeat. The algorithm runs into trouble, however, if none of the input sequences has a nonempty prefix with a nonnegative sum.

In this case, a dual construction allows the processing of the remaining sequences by considering suffixes with nonpositive sums. This will be discussed later.

Lemma 20 *(Shallowest first) Let W be a set of sequences, and suppose that $B(w^k)$ exists and that for each $k' \neq k$, either $B(w^{k'})$ does not exist or $D(B(w^k)) \geq D(B(w^{k'}))$. Let $l = |B(w^k)|$. Then there is some interleaving $I \in \mathcal{I}^*(W)$ such that $M(k,i) = i$ for all $1 \leq i \leq l$, where M is the map associated with I.*

Proof: We first show that for every interleaving $I \in \mathcal{I}(W)$, $D(I) \leq D(B(w^k))$. Let I be an interleaving of W, given by map M. Let k' be the index of the sequence $w^{k'}$ such that $B(w^{k'})$ "finishes first" in I, i.e. $M(k', |B(w^{k'})|) < M(k'', |B(w^{k''})|)$ for all $k'' \neq k'$ such that $B(w^{k''})$ exists. Since each $w^{k''}$, $k'' \neq k'$, contributes a non-positive sum to

$$D(I, M(k', |B(w^{k'})|) = \min_{0 \leq i \leq M(k', |B(w^{k'})|)} \sum_{j=1}^{i} I_j$$

(whether $B(w^{k''})$ exists or not), we have

$$D(I) \leq \min_{0 \leq i \leq |B(w^{k'})|} \sum_{j=1}^{i} w_j^{k'} = D(B(w^{k'})).$$

Since $D(B(w^k)) \geq D(B(w^{k'})$, the claim follows.

Next, we show that any interleaving I, given by map M, that doesn't satisfy the claim of the lemma can be transformed into an interleaving I' (given by a map M') that does, with $D(I') \geq D(I)$. I' is obtained from I by "moving up" the entries in $B(w^k)$ to the beginning of the interleaving, without changing the respective orderings among the entries of $R(w^k)$ and the sequences other than w^k. For $i \leq l$, let $M'(k,i) = i$ and for $i > l$, let $M'(k,i) = M(k,i)$. For each (k', i') such that $k' \neq k$, let $M'(k', i') = M(k', i') + |\{i \leq l : M(k,i) > M(k', i')\}|$. By the preceding argument, the net value $N(I', M'(k,i)) = N(I', i)$ for each $1 \leq i \leq l$ is at least as great as the depth $D(I)$ of the original interleaving. For each $k' \neq k$, each entry $w_{i'}^{k'}$ has a (possibly empty) suffix of $B(w^k)$ moved ahead of it in I'. By Lemma 19, that suffix has a non-negative net, so that $N(I'_1 \ldots I'_{M'(k',i')}) \geq N(I_1 \ldots I_{M(k',i')})$. Thus the overall depth of I' is no smaller than that of I, since at each index of I' there is an index of I with net value at least as small. □

To apply this lemma to the interleaving problem, it is necessary that at least one of the sequences to be interleaved has a non-empty prefix with a non-negative net. When this fails, we use a dual notion. Notice that for a sequence $w = w_1 \dots w_i w_{i+1} \dots w_n$, $N(w_1 \dots w_i) = N(w) - N(w_{i+1} \dots w_n)$. Thus maximizing the minimum prefix sum (of an interleaving) is equivalent to minimizing the maximum suffix sum. This motivates the following:

Definition: For sequence $w = w_1 \dots w_n$, let $B'(w)$ and $R'(w)$ denote the shortest non-empty suffix (if it exists) of w that sums to a non-positive value and the remaining prefix, respectively; that is, $B'(w) = w_{l+1} \dots w_n$ and $R'(w) = w_1 \dots w_l$, where

$$l + 1 = \max_{1 \le i \le n} \{i : N(w_i \dots w_n) \le 0\}$$

if $\{i : N(w_i \dots w_n) \le 0 \,\&\, 1 \le i \le n\} \ne \emptyset$. The *height* $H(w)$ of any sequence $w = w_1 \dots w_n$ is

$$\max_{0 \le i \le n} \sum_{j=i+1}^{n} w_j.$$

A dual argument to Lemma 20 yields the following:

Lemma 21 *(Lowest last) Let W be a set of sequences, and suppose that $B'(w^k)$ exists and that for each $k' \ne k$, either $B'(w^{k'})$ does not exist or $H(B'(w^k)) \le H(B'(w^{k'}))$. Let $l = |R'(w^k)|$. Then there is some interleaving $I \in \mathcal{I}^*(W)$ such that $M(k,i) = (\sum_{j=1}^{m} n_j) - n_k + i$ for all $l + 1 \le i \le n_k$, where M is the map associated with I.*

6.6 The algorithm

Notice that, since every non-empty w is both a non-empty prefix and a non-empty suffix of itself, for every non-empty w either $B(w)$ or $B'(w)$ exists. Notice also that if $B(w)$ does not exist, then removing the suffix $B'(w)$ from w will not change this; i.e. $B(R'(w))$ does not exist. These observations, along with Lemmas 20 and 21, imply that an optimal interleaving of W is obtained by the following algorithm:

Repeat until for each k, $1 \le k \le m$, either w^k is empty or $B(w^k)$ does not exist:

Let k satisfy $D(B(w^k)) \geq D(B(w^{k'}))$ for all k' such that $B(w^{k'})$ exists.
Output $B(w^k)$ and replace w^k with $R(w^k)$.
Initialize a stack S to the empty stack.
Repeat until for each k, $1 \leq k \leq m$, w^k is empty:
Let k satisfy $H(B'(w^k)) \leq H(B'(w^{k'}))$ for all k' such that $w^{k'}$ is not empty.
Push $B'(w^k)$ on S (in reverse order) and replace w^k with $R'(w^k)$.
While S is not empty output $pop(S)$.

A straightforward modification of the algorithm outputs the map M by which the interleaving is obtained from the input set W. The algorithm can be implemented to use $O(n \log m)$ operations (comparisons, additions, and assignments), where n is the sum of the lengths of the input sequences (and equal to the length of the output sequence), and m is the number of input sequences. This is achieved even in the case that each $B(w^k)$ and $B'(w^k)$ is short (length bounded by a constant). A linear scan of each $B(w^k)$ can determine its length and depth (if it exists; if not, a linear scan of all of w^k determines this, and w^k need not be considered again until the second loop is entered.) These records can be stored in a priority queue keyed on the depth. On each iteration of the first loop, a delete maximum operation determines which $B(w^k)$ to output, and which w^k to examine to insert (a description of) the new $B(w^k)$ into the queue. The second loop can be handled similarly. Producing the output has a total cost of $O(n)$. Thus, the most expensive operations are the $O(n)$ insert and delete maximum (or minimum) operations on the priority queue, each with a cost of $O(\log m)$ (see, for example, [3]).

Thus we have the following.

Theorem 22 *The above algorithm finds an optimal interleaving of the m sequences $W = \{w^k = w_1^k \ldots w_{n_k}^k : 1 \leq k \leq m\}$ in time $O(n \log m)$ in the unit cost RAM model.*

6.7 Formalizing the reduction

We return now to the reduction of the prefetching and scheduling problem to the checking and savings account problem. Lemma 20 allows us to assume that the initial non-negative entries of all m sequences (i.e., the numbers of initially uncovered requests) occur first in the interleaving (in an arbitrary order; say in the same order as that in which the input sequences occur). Lemma 20 also allows us to assume that each subsequent pair $(-F, u_i^j)$ corresponding to the i^{th} hole in the j^{th} request sequence occurs consecutively in an interleaving I, since the single positive value u_i^j is a zero-depth prefix. It is thus easy to determine a prefetching schedule that corresponds naturally to an interleaving I with associated map M, such that I_{m+2i} is the number of requests uncovered by the i^{th} fetch; for each prefetching schedule there is a unique such corresponding map specifying an interleaving. In the following, we assume that an interleaving I is known and has been used to determine a prefetching schedule. We claim that $|D(I)|$ is the total stall time of the prefetching schedule.

One direction (the lower bound on total stall time) is easy: since a total of $N(I, m+2i) + iF$ requests are uncovered (initially and by prefetch operations) before time $(i+1)F$, at most $N(I, m+2i) + iF$ requests can be served by time $(i+1)F$. Thus, the stall time accumulated up to time $(i+1)F$ is at least $\max(0, (i+1)F - (iF + N(I, m+2i))) = \max(0, F - N(I, m+2i))$. Unless $m+2i = |I|$, we have that $I_{m+2i+1} = -F$, so that the stall time is at least $\max(0, -N(I, m+2i+1))$. Taking the minimum value over i of $N(I, m+2i+1)$ (and noting that the minimum net is achieved at such an index since I_{m+2i} is positive for each i) yields the lower bound.

We now show that this bound on the stall time can be met. We show by induction on i that at time iF,

1. the number of requests left uncovered and available for servicing between times iF and $(i+1)F$ is $N(I, m+2i) - D(I, m+2i)$, and

2. the accumulated stall time is $|D(I, m+2i)|$.

The basis ($i = 0$) is trivial. For the induction, assume the hypothesis is true for i.

Case 1: $D(I, m+2i+2) = D(I, m+2i)$. In this case, $N(I, m+2i+1) \geq D(I, m+2i)$ so that $N(I, m+2i) - D(I, m+2i) \geq F$, since $I_{m+2i+1} = -F$. Thus, by the induction hypothesis, there are at least F requests uncovered at time iF, and no further stalling is incurred between times iF and $(i+1)F$. F requests are served between times iF and $(i+1)F$, the accumulated stall time is (unchanged) $|D(I, m+2i)| = |D(I, m+2i+2)|$, and the number of requests left uncovered is $N(I, m+2i) - D(I, m+2i) - F + I_{m+2i+2} = N(I, m+2i+2) - D(I, m+2i+2)$ as needed.

Case 2: $D(I, m+2i+2) < D(I, m+2i)$. In this case, $N(I, m+2i) - D(I, m+2i) < F$, and the number of additional stall steps incurred is $F - (N(I, m+2i) - D(I, m+2i)) = D(I, m+2i) - N(I, m+2i+1) = D(I, m+2i) - D(I, m+2i+2)$ so that the total is $|D(I, m+2i+2)|$. The number of requests left uncovered at time $(i+1)F$ is $I_{m+2i+2} = N(I, m+2i+2) - N(I, m+2i+1) = N(I, m+2i+2) - D(I, m+2i+2)$ as needed.

Chapter 7

HARDNESS OF ORDERING REQUEST SEQUENCES TO MINIMIZE CACHE MISSES

In this chapter, we consider a generalization of the classic paging problem raised by Philbin *et al.* [37]. Suppose we are given m distinct sequences of references, and must process the sequences in succession. That is, each sequence must be served in its entirety before serving another sequence. However, the ordering of the individual sequences is arbitrary. We show that minimizing the number of cache misses is NP-hard for direct-mapped, set-associative, and fully-associative caches.

7.1 Thread scheduling for improved locality

Philbin *et al.* describe the use of independent threads of control for increasing a program's locality of reference. The threads are scheduled at run-time, allowing the use of information that is not available, such as the values of pointers, at compile-time in order to apply optimizations that improve cache performance. Even though there is a run-time cost, they demonstrate that this technique can improve performance for some applications by reducing the number of processor cache misses. Philbin *et al.* use a clever heuristic to cluster threads based on the addresses of the data they reference. The addresses are passed to the thread scheduler at run-time as threads are created.

7.2 Hardness of the thread scheduling problem

We formalize the thread scheduling problem as follows. Let B denote a set of blocks that reside on a backing store. A *reference sequence*, or *request sequence*, is an ordered sequence of references $R^i = r_1^i, r_2^i, \ldots r_{|R^i|}^i$, where each $r_j^i \in B$. There are m separate reference sequences $R^1, \ldots, R^m$. There is a cache of size K that contains at most

K blocks in B at any time. As mentioned, the references in each sequence R^i must be served in order and consecutively, but there is no restriction on the ordering of the sequences. A block must be present in the cache when a reference to it is served. The goal is to construct, on input request sequences $\{R^i\}$, an ordering of the sequences such that the number of cache misses is minimized. (The number of misses is easily determined using Belady's algorithm [4] for an associative cache, and by an even simpler method for a direct-mapped cache, once the individual sequences are ordered).

We reduce the Directed Hamiltonian Path problem to the sequence ordering problem. This problem remains NP-complete for graphs in which no vertex is incident to more than three edges [14]. A simple transformation allows us to assume that no vertex has outdegree or indegree greater than two. This will allow us to take the cache size to be a constant ($K \geq 4$ is enough) rather than to depend on the input size. The transformation is as follows: replace any vertex v with outdegree three and neighbors w_1, w_2, and w_3 by a directed cycle on new vertices v_1, v_2 and v_3 plus the edges (v_1, w_1), (v_2, w_2) and (v_3, w_3). v must have indegree zero and must hence be the first vertex on any Hamiltonian path in the original graph. In the new graph, any Hamiltonian path must first traverse two of the three edges on the cycle, then exit the cycle to visit w_1, w_2, or w_3. Thus every Hamiltonian path in the new graph corresponds to a Hamiltonian path in the original graph. Vertices of indegree three are handled similarly.

First we prove that the sequence ordering problem is NP-hard for a direct-mapped cache. We will refer to the cache location to which a block is mapped as the *color* of the block. The colors used are 0, 1, 2, and 3, corresponding to the (first) four cache locations. Let $G = (V, E)$ be a directed graph with the specified degree restrictions. We define a set of blocks B that contains one block $b_{i,j}$ corresponding to each directed edge $(v_i, v_j) \in E$. Let each vertex arbitrarily assign different colors to each of its (at most two) in-edges from the colors 0 and 1, and to each of its (at most two) out-edges from the colors 0 and 2. Let the sum of the colors assigned by an edge's two incident vertices be the color of the edge. Each edge's color is assigned to the corresponding block, and thus determines to which cache location the block maps. It is easy to see that no two edges into or out of the same vertex have the same color, so that the corresponding blocks will not conflict (map to the same cache location). In addition to the blocks corresponding to the edges of G, for each vertex $v_i \in V$, B contains four "private blocks" $p_{i,0} \ldots p_{i,3}$, where $p_{i,j}$ is colored j.

For each vertex $v_i \in V$, we construct a request sequence R^i. R^i begins with an initial subsequence consisting of one request for each block $b_{j,i}$ corresponding to an edge into v_i (in either order, if there are two). These are follcwed by a middle subsequence $p_{i,0}p_{i,1}p_{i,2}p_{i,3}$. Finally, R^i contains a final subsequence consisting of one request for each block $b_{i,j}$ corresponding to an edge out of v_i (in either order, if there are two).

Each sequence R^i will suffer 4 misses to bring its private blocks into the cache. It is easy to see that during the servicing of each R^i, R^i's private blocks will flush any shared blocks from the cache. Thus, each R^i will also suffer $outdegree(v_i)$ misses to bring its final subsequence (corresponding to the out-edges of v_i) into the cache. These blocks will remain in the cache at the end of the servicing of R^i, since they do not conflict. Each R^i will suffer either $indegree(i)$ or $indegree(i) - 1$ misses to bring its initial subsequence into the cache, depending on whether the previously served sequence corresponds to a predecessor of v_i in G. This is true regardless of the order of the requests in R^i's initial subsequence, since if it contains two blocks, they do not conflict. Thus G has a Hamiltonian path if and only if the optimal ordering of the sequences suffers only

$$\sum_{v_i \in V} outdegree(v_i) + \sum_{v_i \in V} indegree(v_i) + 4|V| - (|V| - 1) = 2|E| + 3|V| + 1$$

cache misses.

The fully-associative cache case requires a bit more work to ensure that the private blocks flush all shared blocks from the cache. Let K be the size of the cache; we will need K distinct private blocks for each vertex. The middle subsequence of each R^i is replaced by $p_{i,0} \ldots p_{i,K-1} p_{i,0} \ldots p_{i,K-1}$. As before, each of the private blocks will cause a miss. The second reference to each $p_{i,j}$ ensures that none of the private blocks will be evicted before all K of them have been brought into the cache,[1] by Belady's [4] longest forward distance page replacement rule, since each must be served again before any other blocks. Thus, after the first pass through R^i's private blocks, all K

[1] To be precise, we can assume without loss of generality that this holds for any optimal schedule.

of them reside in the cache and all shared blocks are missing. The number of misses that corresponds to a Hamiltonian path becomes

$$2|E| + (K-1)|V| + 1.$$

For a t-way-associative cache with $t \geq 4$, we use the same proof as for a fully-associative cache (replacing K with t, and simply arranging that all blocks map to the same set). For $2 \leq t \leq 3$, a straightforward combination of the fully-associative and direct-mapped methods can be used. Blocks corresponding to edges are colored as in the direct-mapped case; t private blocks of each color, requested twice each as in the fully-associative case, assure the cache is flushed of all shared blocks. Again, the number of misses that corresponds to a Hamiltonian path must be adjusted accordingly.

Simple modifications also show the problem to be NP-hard if we assume a fixed block-replacement policy such as LRU rather than optimal offline replacement in the associative cache cases.

Chapter 8

CONCLUSION AND DIRECTIONS FOR FURTHER RESEARCH

In Chapter 3 we presented a theoretical analysis of algorithms for the parallel prefetching and caching problem. We showed that *reverse aggressive* is guaranteed to find a solution within a factor near one of optimal, and that all the other algorithms we considered can perform much worse in the worst case. Chapter 4 presented 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 thesis leaves unresolved several important issues related to parallel prefetching and caching. The problem of finding an optimal algorithm in the abstract theoretical model with running time polynomial in both the input size and the cache size remains open, even for a single disk. 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 evaluated the performance of the algorithms in cases of imperfect lookahead. If the lookahead information is not quite perfect but highly accurate, it is reasonable to expect the deviations from the predictions of our analysis and experiments to be small. A very interesting direction is to extend these results to the case in which only probabilistic lookahead information is available. How much confidence is needed before prefetching yields an expected payoff?

This work demonstrates that a file system can effectively take advantage of accurate lookahead information. An important research direction is to determine methods by which applications can easily provide such hints.

In Chapter 5, we presented an efficient algorithm to determine whether a request sequence can be served with zero stall time in the single disk prefetching and caching model. The algorithm produces a schedule to serve a sequence without stalling, if possible. The algorithm can be applied to problems in which the cache is initially empty, and in this case, determines whether the sequence can be served without stalling more than is necessary to fill the cache starting from the cold cache initial state.

In Chapter 6, we considered an abstract combinatorial problem, "sequence interleaving," derived from the integrated prefetching, caching, and processor scheduling problem. A simple and efficient algorithm was given for the sequence interleaving problem. The sequence interleaving problem corresponds to a simplification of the integrated scheduling problem, which appears to be difficult. However, there is reason to believe that a solution to the simplified problem will perform well in practice, as discussed in Section 6.1. A direction for future work is the testing of this hypothesis by comparing the algorithm to existing approaches, using simulation based on traces of real programs' resource demands and/or development of a prototype system that uses the algorithm.

In Chapter 7, a problem of increasing a program's locality of reference by scheduling independent threads of control was described, and a proof of NP-hardness was given. The development of approximation algorithms for this problem and/or lower bounds on its approximability are very interesting problems, as is that of determining bounds on the quality of the approximation produced by the heuristic of Philbin *et al.* [37].

Appendix A

SIMULATION DATA

A.1 Performance data: baseline measurements

This section contains the raw simulation data for the baseline parameters as described in Chapter 4: the prefetch horizon of *fixed horizon* is 62, *aggressive*'s batch size is set according to table 4.1, *reverse aggressive*'s fetch time estimate F and batch size are chosen to minimize its elapsed time, and Forestall's fetch time estimate F' is determined dynamically as described in section 4.6.

Table A.1: Performance on the dinero trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	4771	4771	4771	4771	4771	4771
driver time (sec)	2.3855	2.3855	2.3855	2.3855	2.3855	2.3855
stall time (sec)	0.027	0.009	0.009	0.009	0.009	0.009
elapsed time (sec)	105.951	105.933	105.933	105.933	105.933	105.933
average fetch time (msec)	3.156	3.178	3.233	3.258	3.274	3.319
average disk utilization	0.14	0.072	0.049	0.037	0.029	0.025
Aggressive						
fetches	8812	8812	8823	8815	8812	8816
driver time (sec)	4.406	4.406	4.4115	4.4075	4.406	4.408
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	107.944	107.95	107.946	107.944	107.947
average fetch time (msec)	3.141	3.146	3.174	3.176	3.188	3.203
average disk utilization	0.26	0.13	0.086	0.065	0.052	0.044
Reverse Aggressive						
fetches	4731	4764	4829	4830	4914	5018
driver time (sec)	2.3655	2.382	2.4145	2.415	2.457	2.509
stall time (sec)	0.023	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	105.927	105.941	105.972	105.97	106.01	106.06
average fetch time (msec)	3.31	3.36	3.366	3.437	3.494	3.288
average disk utilization	0.15	0.076	0.051	0.039	0.032	0.026
Forestall						
fetches	4753	4753	4753	4753	4753	4753
driver time (sec)	2.3765	2.3765	2.3765	2.3765	2.3765	2.3765
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	106.06	105.915	105.915	105.915	105.915	105.916
average fetch time (msec)	3.183	3.196	3.253	3.272	3.298	3.324
average disk utilization	0.14	0.072	0.049	0.037	0.03	0.025

Table A.2: Performance on the cscope1 trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	4953	4953	4953	4953	4953	4953
driver time (sec)	2.4765	2.4765	2.4765	2.4765	2.4765	2.4765
stall time (sec)	3.131	0.013	0.013	0.013	0.013	0.013
elapsed time (sec)	30.542	27.424	27.424	27.424	27.424	27.424
average fetch time (msec)	3.53	3.239	3.248	3.286	3.317	3.355
average disk utilization	0.57	0.29	0.2	0.15	0.12	0.1
Aggressive						
fetches	6931	8570	8672	8678	8621	8576
driver time (sec)	3.4655	4.285	4.336	4.339	4.3105	4.288
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.219	29.27	29.273	29.245	29.223
average fetch time (msec)	3.758	3.361	3.429	3.365	3.39	3.356
average disk utilization	0.89	0.49	0.34	0.25	0.2	0.16
Reverse Aggressive						
fetches	5349	4995	5024	5093	5132	5135
driver time (sec)	2.6745	2.4975	2.512	2.5465	2.566	2.5675
stall time (sec)	1.312	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	28.921	27.453	27.465	27.498	27.515	27.515
average fetch time (msec)	3.622	3.344	3.376	3.409	3.396	3.618
average disk utilization	0.67	0.3	0.21	0.16	0.13	0.11
Forestall						
fetches	5210	4970	4953	4953	4953	4953
driver time (sec)	2.605	2.485	2.4765	2.4765	2.4765	2.4765
stall time (sec)	1.266	0	0	0	0	0.001
elapsed time (sec)	28.805	27.419	27.411	27.411	27.411	27.412
average fetch time (msec)	3.794	3.334	3.276	3.295	3.326	3.342
average disk utilization	0.69	0.3	0.2	0.15	0.12	0.1

Table A.3: Performance on the cscope2 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	32.802	22.261	14.616	9.04	5.921	3.905	2.488	1.347	1.016	0.371	0.133
elapsed time (sec)	72.894	62.353	54.708	49.132	46.013	43.997	42.58	41.439	41.108	40.463	40.225
average fetch time (msec)	9.469	15.009	17.309	17.993	18.463	18.921	18.894	19.083	19.216	19.217	19.542
average disk utilization	0.77	0.72	0.63	0.55	0.48	0.43	0.38	0.34	0.28	0.24	0.18
Aggressive											
fetches	6318	6592	8208	8956	10299	11014	11587	11717	11619	11102	10662
driver time (sec)	3.159	3.296	4.104	4.478	5.1495	5.507	5.7935	5.8585	5.8095	5.551	5.331
stall time (sec)	15.858	5.597	1.798	0	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	56.126	46.002	43.011	41.587	42.259	42.617	42.903	42.977	42.924	42.661	42.44
average fetch time (msec)	8.773	13.256	14.354	16.514	17.138	17.683	17.722	17.551	17.224	17.65	18.201
average disk utilization	0.99	0.95	0.91	0.89	0.84	0.76	0.68	0.6	0.47	0.38	0.29
Reverse Aggressive											
fetches	6359	7320	6837	6290	6124	6071	6085	6115	6131	6177	6237
driver time (sec)	3.1795	3.66	3.4185	3.145	3.062	3.0355	3.0425	3.0575	3.0655	3.0885	3.1185
stall time (sec)	17.966	6.057	0.978	0	0.005	0.013	0.011	0.009	0.005	0.016	0.008
elapsed time (sec)	58.255	46.826	41.506	40.254	40.176	40.158	40.163	40.176	40.18	40.214	40.236
average fetch time (msec)	8.173	11.43	13.428	16.847	17.651	17.939	18.64	18.616	19.054	19.133	19.285
average disk utilization	0.89	0.89	0.74	0.66	0.54	0.45	0.4	0.35	0.29	0.24	0.19
Forestall											
fetches	6318	6467	7217	7239	7715	7387	7355	7187	7086	6853	6476
driver time (sec)	3.159	3.2335	3.6085	3.6195	3.8575	3.6935	3.6775	3.5935	3.543	3.4265	3.238
stall time (sec)	15.858	5.677	1.798	0	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	56.126	46.02	42.516	40.729	40.967	40.804	40.787	40.712	40.657	40.537	40.347
average fetch time (msec)	8.773	13.251	14.466	16.675	16.97	18.158	18.274	18.821	19.133	19.123	19.23
average disk utilization	0.99	0.93	0.82	0.74	0.64	0.55	0.47	0.42	0.33	0.27	0.19

Table A.4: Performance on the cscope3 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739
driver time (sec)	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	28.459	12.906	7.046	2.961	1.669	0.762	0.221	0.164	0.152	0.014	0.014
elapsed time (sec)	108.429	92.876	87.016	82.931	81.639	80.732	80.191	80.134	80.122	79.984	79.984
average fetch time (msec)	7.843	11.914	14.814	16.147	16.993	17.482	17.906	18.178	18.671	18.875	19.108
average disk utilization	0.85	0.75	0.67	0.57	0.49	0.42	0.37	0.33	0.27	0.23	0.18
Aggressive											
fetches	12092	13572	15938	16740	17713	18081	17894	17577	16917	16542	16314
driver time (sec)	6.046	6.786	7.969	8.37	8.8565	9.0405	8.947	8.7885	8.4585	8.271	8.157
stall time (sec)	13.943	2.862	0.64	0.052	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	94.09	83.749	82.71	82.523	82.957	83.142	83.048	82.898	82.564	82.373	82.258
average fetch time (msec)	7.741	11.597	14.215	15.92	16.553	16.568	16.711	16.905	17.605	17.966	18.49
average disk utilization	0.99	0.94	0.91	0.81	0.71	0.6	0.51	0.45	0.36	0.3	0.23
Reverse Aggressive											
fetches	12228	12814	12501	12033	11880	11837	11852	11883	11919	11954	12004
driver time (sec)	6.114	6.407	6.2505	6.0165	5.94	5.9185	5.926	5.9415	5.9595	5.977	6.002
stall time (sec)	23.85	3.531	0.66	0.407	0.006	0.013	0.011	0.009	0.005	0.016	0.008
elapsed time (sec)	104.065	84.039	81.01[illegible]	80.524	80.047	80.032	80.038	80.051	80.065	80.094	80.111
average fetch time (msec)	7.763	10.787	14.095	15.97	16.612	17.358	17.844	18.127	18.482	18.749	19.025
average disk utilization	0.91	0.82	0.73	0.6	0.49	0.43	0.38	0.34	0.28	0.23	0.18
Forestall											
fetches	12054	13115	14217	13969	14125	13878	13846	13589	13322	13048	12536
driver time (sec)	6.027	6.5575	7.1085	6.9845	7.0625	6.939	6.923	6.7945	6.661	6.524	6.268
stall time (sec)	14.273	2.863	0.64	0.052	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	94.401	83.521	81.849	81.137	81.163	81.041	81.024	80.904	80.767	80.626	80.369
average fetch time (msec)	7.731	11.603	13.616	15.745	16.183	17.246	17.648	18.477	18.669	18.812	19.035
average disk utilization	0.99	0.91	0.79	0.68	0.56	0.49	0.43	0.39	0.31	0.25	0.19

Table A.5: Performance on the glimpse trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493
driver time (sec)	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465
stall time (sec)	65.619	31.046	20.054	14.029	10.381	7.886	5.702	4.769	2.809	1.404	0.722
elapsed time (sec)	107.582	73.009	62.017	55.992	52.344	49.849	47.665	46.732	44.772	43.367	42.685
average fetch time (msec)	13.424	15.145	16.192	17.244	18.068	18.33	18.452	18.642	18.555	18.571	18.743
average disk utilization	0.81	0.67	0.57	0.5	0.45	0.4	0.36	0.32	0.27	0.23	0.18
Aggressive											
fetches	6690	6888	7287	7551	8908	9376	10423	10992	12009	11530	11315
driver time (sec)	3.345	3.444	3.6435	3.7755	4.454	4.688	5.2115	5.496	6.0045	5.765	5.6575
stall time (sec)	54.58	18.58	6.384	2.495	0.826	0.035	0	0.009	0.005	0.001	0
elapsed time (sec)	96.641	60.74	48.744	44.987	43.996	43.439	43.928	44.221	44.726	44.482	44.374
average fetch time (msec)	12.889	14.259	14.645	16.247	15.973	16.836	16.79	16.896	16.137	15.917	16.198
average disk utilization	0.89	0.81	0.73	0.68	0.65	0.61	0.57	0.52	0.43	0.34	0.26
Reverse Aggressive											
fetches	6712	7179	7630	8141	7619	6803	6656	6709	6750	6822	6978
driver time (sec)	3.356	3.5895	3.815	4.0705	3.8095	3.4015	3.328	3.3545	3.375	3.411	3.489
stall time (sec)	52.011	15.928	4.971	0.495	0	0	0.011	0.009	0.005	0.006	0
elapsed time (sec)	94.083	58.234	47.502	43.282	42.526	42.118	42.055	42.08	42.096	42.133	42.205
average fetch time (msec)	12.745	13.46	13.793	13.877	14.73	17.016	18.321	18.37	18.541	18.514	18.406
average disk utilization	0.91	0.83	0.74	0.65	0.53	0.46	0.41	0.37	0.3	0.25	0.19
Forestall											
fetches	6610	6617	6945	6905	7033	6937	7113	7093	7125	7089	6941
driver time (sec)	3.305	3.3085	3.4725	3.4525	3.5165	3.4685	3.5565	3.5465	3.5625	3.5445	3.4705
stall time (sec)	54.886	18.833	6.58	2.906	1.397	0.099	0	0.009	0.005	0.001	0
elapsed time (sec)	96.907	60.858	48.769	45.075	43.63	42.284	42.273	42.272	42.284	42.262	42.187
average fetch time (msec)	13.001	14.378	14.797	16.382	16.651	17.627	17.557	18.099	18.015	18.055	18.273
average disk utilization	0.89	0.78	0.7	0.63	0.54	0.48	0.42	0.38	0.3	0.25	0.19

Table A.6: Performance on the ld trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horison											
fetches	2904	2904	2904	2904	2904	2904	2904	2904	2904	2904	2904
driver time (sec)	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452
stall time (sec)	15.281	7.297	4.696	3.043	2.086	1.565	1.212	1.041	0.599	0.416	0.269
elapsed time (sec)	24.898	16.914	14.313	12.66	11.703	11.182	10.829	10.658	10.216	10.033	9.886
average fetch time (msec)	8.368	10.94	13.299	15.031	16.214	16.93	17.502	17.657	18.467	18.945	19.2
average disk utilization	0.98	0.94	0.9	0.86	0.8	0.73	0.67	0.6	0.52	0.46	0.35
Aggressive											
fetches	2981	2982	3137	3102	3310	3505	3734	3779	4091	4285	4651
driver time (sec)	1.4905	1.491	1.5685	1.551	1.655	1.7525	1.867	1.8895	2.0455	2.1425	2.3255
stall time (sec)	15.245	6.329	3.433	2.052	0.579	0.265	0.023	0.009	0.005	0.001	0
elapsed time (sec)	24.9	15.985	13.166	11.768	10.399	10.182	10.055	10.063	10.215	10.308	10.49
average fetch time (msec)	8.248	10.583	12.037	14.199	14.932	15.958	16.444	17.175	17.62	18.017	18.261
average disk utilization	0.99	0.99	0.96	0.94	0.95	0.92	0.87	0.81	0.71	0.62	0.51
Reverse Aggressive											
fetches	3041	3079	3202	3312	3161	3037	3103	3000	2953	3004	3008
driver time (sec)	1.5205	1.5395	1.601	1.656	1.5805	1.5185	1.5515	1.5	1.4765	1.502	1.504
stall time (sec)	14.662	6.217	3.233	1.704	0.879	0.618	0.211	0.151	0.035	0.016	0.008
elapsed time (sec)	24.347	15.921	12.999	11.525	10.624	10.301	9.927	9.816	9.676	9.683	9.677
average fetch time (msec)	7.932	10.036	11.585	13.254	14.43	16.016	16.269	17.39	18.558	18.995	18.992
average disk utilization	0.99	0.97	0.95	0.95	0.86	0.79	0.73	0.66	0.57	0.49	0.37
Forestall											
fetches	2981	2982	3137	3102	3310	3505	3734	3799	3896	3799	3147
driver time (sec)	1.4905	1.491	1.5685	1.551	1.655	1.7525	1.867	1.8995	1.948	1.8995	1.5735
stall time (sec)	15.245	6.329	3.433	2.052	0.579	0.265	0.023	0.013	0.005	0.001	0
elapsed time (sec)	24.9	15.985	13.166	11.768	10.399	10.182	10.055	10.077	10.118	10.065	9.738
average fetch time (msec)	8.248	10.583	12.037	14.199	14.932	15.958	16.442	17.253	18.126	18.582	18.983
average disk utilization	0.99	0.99	0.96	0.94	0.95	0.92	0.87	0.81	0.7	0.58	0.38

Table A.7: Performance on the postgres-join trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	3856	3856	3856	3856	3856	3856
driver time (sec)	1.928	1.928	1.928	1.928	1.928	1.928
stall time (sec)	4.723	0.04	0.017	0.017	0.017	0.017
elapsed time (sec)	85.867	81.184	81.161	81.161	81.161	81.161
average fetch time (msec)	17.228	18.029	18.039	18.299	18.344	18.094
average disk utilization	0.77	0.43	0.29	0.22	0.17	0.14
Aggressive						
fetches	4698	5836	6225	6156	6047	5919
driver time (sec)	2.349	2.918	3.1125	3.078	3.0235	2.9595
stall time (sec)	3.994	0.152	0.258	0	0	0.001
elapsed time (sec)	85.559	82.286	82.586	82.294	82.239	82.176
average fetch time (msec)	15.032	16.576	15.929	16.578	16.706	17.102
average disk utilization	0.83	0.59	0.4	0.31	0.25	0.21
Reverse Aggressive						
fetches	3987	3853	3859	3873	3879	3892
driver time (sec)	1.9935	1.9265	1.9295	1.9365	1.9395	1.946
stall time (sec)	3.775	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	84.984	81.163	81.164	81.169	81.17	81.175
average fetch time (msec)	15.776	18.055	17.964	18.204	18.262	17.934
average disk utilization	0.74	0.43	0.28	0.22	0.17	0.14
Forestall						
fetches	4694	4207	3929	3857	3855	3856
driver time (sec)	2.347	2.1035	1.9645	1.9285	1.9275	1.928
stall time (sec)	3.994	0.153	0.258	0	0	0.001
elapsed time (sec)	85.557	81.472	81.438	81.144	81.143	81.145
average fetch time (msec)	15.034	15.022	15.525	17.291	17.45	17.692
average disk utilization	0.82	0.39	0.25	0.21	0.17	0.14

Table A.8: Performance on the postgres-select trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	32.37	12.647	5.943	3.154	1.402	0.581	0.476	0.073	0.034	0.018	0.018
elapsed time (sec)	45.39	25.667	18.963	16.174	14.422	13.601	13.496	13.093	13.054	13.038	13.038
average fetch time (msec)	14.368	14.906	15.044	15.13	15.347	15.413	15.437	15.411	15.278	15.356	15.071
average disk utilization	0.98	0.9	0.82	0.72	0.66	0.58	0.5	0.45	0.36	0.3	0.22
Aggressive											
fetches	3085	3085	3085	3085	3286	3317	3826	3937	3902	3852	3731
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.643	1.6585	1.913	1.9685	1.951	1.926	1.8655
stall time (sec)	30.691	10.772	3.517	0.844	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.792	16.537	13.864	13.121	13.137	13.391	13.455	13.434	13.405	13.343
average fetch time (msec)	13.985	14.173	13.95	14.55	13.923	15.036	15.221	15.274	14.797	14.8	15.155
average disk utilization	0.99	0.92	0.87	0.81	0.7	0.63	0.62	0.56	0.43	0.35	0.26
Reverse Aggressive											
fetches	3106	3106	3318	3110	3109	3108	3112	3122	3116	3122	3124
driver time (sec)	1.553	1.553	1.659	1.555	1.5545	1.554	1.556	1.561	1.558	1.561	1.562
stall time (sec)	28.956	8.461	2.66	0.125	0	0.001	0	0	0	0	0.002
elapsed time (sec)	41.987	21.492	15.797	13.158	13.032	13.033	13.034	13.039	13.036	13.039	13.042
average fetch time (msec)	13.248	12.704	12.127	13.581	14.394	15.051	14.803	14.049	14.618	14.024	14.143
average disk utilization	0.98	0.92	0.85	0.8	0.69	0.6	0.5	0.42	0.35	0.28	0.21
Forestall											
fetches	3085	3085	3085	3085	3085	3305	3797	3795	3399	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.6525	1.8985	1.8975	1.6995	1.5425	1.5425
stall time (sec)	30.691	10.791	3.517	0.844	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.811	16.537	13.864	13.02	13.131	13.376	13.384	13.182	13.021	13.02
average fetch time (msec)	13.985	14.154	13.933	14.524	14.392	15.056	15.242	15.249	15.2	15.086	15.032
average disk utilization	0.99	0.92	0.87	0.81	0.68	0.63	0.62	0.54	0.39	0.3	0.22

Table A.9: Performance on the synth trace.

Disks	1	2	3	4
Fixed Horizon				
fetches	38000	38000	38000	38000
driver time (sec)	19	19	19	19
stall time (sec)	82.583	12.044	0	0
elapsed time (sec)	201.439	130.9	118.856	118.856
average fetch time (msec)	3.748	3.776	3.229	3.214
average disk utilization	0.71	0.55	0.34	0.26
Aggressive				
fetches	39240	41902	100994	100548
driver time (sec)	19.62	20.951	50.497	50.274
stall time (sec)	36.37	0.933	0.015	0.015
elapsed time (sec)	155.846	121.74	150.368	150.145
average fetch time (msec)	3.965	5.647	3.37	3.164
average disk utilization	1	0.97	0.75	0.53
Reverse Aggressive				
fetches	39265	42000	37907	38148
driver time (sec)	19.6325	21	18.9535	19.074
stall time (sec)	41.599	2.765	0.014	0.015
elapsed time (sec)	161.088	123.621	118.824	118.945
average fetch time (msec)	3.928	3.907	3.762	3.958
average disk utilization	0.96	0.66	0.4	0.32
Forestall				
fetches	39240	38900	39838	38000
driver time (sec)	19.62	19.45	19.919	19
stall time (sec)	36.37	1.232	0.016	0
elapsed time (sec)	155.846	120.538	119.791	118.856
average fetch time (msec)	3.965	4.895	4.843	3.218
average disk utilization	1	0.79	0.54	0.26

Table A.10: Performance on the xds trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	5900	5900	5900	5900	5900	5900
driver time (sec)	2.95	2.95	2.95	2.95	2.95	2.95
stall time (sec)	32.582	4.964	3.219	1.138	0.474	0.094
elapsed time (sec)	65.611	37.993	36.248	34.167	33.503	33.123
average fetch time (msec)	10.74	7.758	14.065	10.106	15.07	10.869
average disk utilization	0.97	0.6	0.76	0.44	0.53	0.32
Aggressive						
fetches	5925	7778	6563	9831	8312	10215
driver time (sec)	2.9625	3.889	3.2815	4.9155	4.156	5.1075
stall time (sec)	30.667	0.337	0.356	0.129	0.133	0.055
elapsed time (sec)	63.708	34.305	33.716	35.123	34.368	35.241
average fetch time (msec)	10.711	7.496	14.101	9.801	15.454	10.711
average disk utilization	1	0.85	0.91	0.69	0.75	0.52
Reverse Aggressive						
fetches	5892	5989	5927	6001	5893	6017
driver time (sec)	2.946	2.9945	2.9635	3.0005	2.9465	3.0085
stall time (sec)	31.155	0.275	0.528	0.046	0.017	0.018
elapsed time (sec)	64.18	33.348	33.57	33.125	33.042	33.105
average fetch time (msec)	10.79	7.732	14.092	9.864	14.883	10.173
average disk utilization	0.99	0.69	0.83	0.45	0.53	0.31
Forestall						
fetches	5925	6929	6553	7451	7882	7032
driver time (sec)	2.9625	3.4645	3.2765	3.7255	3.941	3.516
stall time (sec)	30.667	0.337	0.356	0.129	0.133	0.055
elapsed time (sec)	63.708	33.88	33.711	33.933	34.153	33.65
average fetch time (msec)	10.711	7.559	14.082	9.945	15.53	10.7
average disk utilization	1	0.77	0.91	0.55	0.72	0.37

A.2 Performance data: FCFS

This section contains the data for the baseline parameters as in the previous section, but with FCFS disk head scheduling rather than CSCAN.

Table A.11: Performance on the dinero trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	4771	4771	4771	4771	4771	4771
driver time (sec)	2.3855	2.3855	2.3855	2.3855	2.3855	2.3855
stall time (sec)	0.009	0.009	0.009	0.009	0.009	0.009
elapsed time (sec)	105.933	105.933	105.933	105.933	105.933	105.933
average fetch time (msec)	3.153	3.181	3.218	3.247	3.26	3.314
average disk utilization	0.14	0.072	0.048	0.037	0.029	0.025
Aggressive						
fetches	8812	8812	8812	8814	8812	8814
driver time (sec)	4.406	4.406	4.406	4.407	4.406	4.407
stall time (sec)	0	0	0	0	0	0.001
elapsed time (sec)	107.944	107.944	107.944	107.945	107.944	107.946
average fetch time (msec)	3.148	3.144	3.166	3.169	3.178	3.194
average disk utilization	0.26	0.13	0.086	0.065	0.052	0.043
Reverse Aggressive						
fetches	4731	4764	4829	4830	4914	5018
driver time (sec)	2.3655	2.382	2.4145	2.415	2.457	2.509
stall time (sec)	0.023	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	105.927	105.941	105.972	105.97	106.01	106.06
average fetch time (msec)	3.311	3.356	3.352	3.431	3.483	3.284
average disk utilization	0.15	0.075	0.051	0.039	0.032	0.026

Table A.12: Performance on the cscopel trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	4953	4953	4953	4953	4953	4953
driver time (sec)	2.4765	2.4765	2.4765	2.4765	2.4765	2.4765
stall time (sec)	3.131	0.013	0.013	0.013	0.013	0.013
elapsed time (sec)	30.542	27.424	27.424	27.424	27.424	27.424
average fetch time (msec)	3.533	3.245	3.247	3.277	3.305	3.335
average disk utilization	0.57	0.29	0.2	0.15	0.12	0.1
Aggressive						
fetches	6778	8582	8606	8685	8621	8576
driver time (sec)	3.389	4.291	4.303	4.3425	4.3105	4.288
stall time (sec)	0.609	0	0	0	0	0.001
elapsed time (sec)	28.932	29.225	29.237	29.277	29.245	29.223
average fetch time (msec)	3.808	3.405	3.373	3.358	3.377	3.341
average disk utilization	0.89	0.5	0.33	0.25	0.2	0.16
Reverse Aggressive						
fetches	5349	4995	5024	5093	5132	5135
driver time (sec)	2.6745	2.4975	2.512	2.5465	2.566	2.5675
stall time (sec)	1.162	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	28.771	27.453	27.465	27.498	27.515	27.515
average fetch time (msec)	3.656	3.342	3.366	3.4	3.382	3.619
average disk utilization	0.68	0.3	0.21	0.16	0.13	0.11

Table A.13: Performance on the cscope2 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	35.261	24.715	15.262	7.728	4.287	2.408	1.273	0.741	0.555	0.087	0.03
elapsed time (sec)	75.353	64.807	55.354	47.82	44.379	42.5	41.365	40.833	40.647	40.179	40.122
average fetch time (msec)	9.887	15.936	17.771	18.352	18.473	18.93	18.913	19.165	19.128	19.29	19.404
average disk utilization	0.78	0.73	0.64	0.57	0.5	0.44	0.39	0.35	0.28	0.24	0.18
Aggressive											
fetches	6196	6324	7302	8450	9933	10777	11475	11707	11529	11102	10662
driver time (sec)	3.098	3.162	3.651	4.225	4.9665	5.3885	5.7375	5.8535	5.7645	5.551	5.331
stall time (sec)	17.951	8.281	3.024	0	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	58.158	48.552	43.784	41.334	42.076	42.499	42.847	42.972	42.879	42.661	42.44
average fetch time (msec)	9.358	14.89	16.761	17.499	17.804	17.917	17.767	17.571	17.348	17.734	18.257
average disk utilization	1	0.97	0.93	0.89	0.84	0.76	0.68	0.6	0.47	0.38	0.29
Reverse Aggressive											
fetches	6359	7320	6837	6290	6124	6071	6085	6115	6131	6177	6237
driver time (sec)	3.1795	3.66	3.4185	3.145	3.062	3.0355	3.0425	3.0575	3.0655	3.0885	3.1185
stall time (sec)	19.611	12.595	3.118	0	0.017	0.013	0.011	0.009	0.005	0.016	0.008
elapsed time (sec)	59.9	53.364	43.646	40.254	40.188	40.158	40.163	40.176	40.18	40.214	40.236
average fetch time (msec)	8.869	13.946	16.125	17.786	18.115	18.048	18.615	18.678	19.186	19.328	19.283
average disk utilization	0.94	0.96	0.84	0.69	0.55	0.45	0.4	0.36	0.29	0.25	0.19

Table A.14: Performance on the cscope3 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739
driver time (sec)	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	31.757	13.743	6.463	2.014	0.749	0.233	0.2	0.014	0.095	0.014	0.014
elapsed time (sec)	111.727	93.713	86.433	81.984	80.719	80.203	80.17	79.984	80.065	79.984	79.984
average fetch time (msec)	8.184	12.268	15.013	16.108	17.015	17.456	17.891	18.277	18.654	18.88	19.1
average disk utilization	0.86	0.77	0.68	0.58	0.49	0.43	0.37	0.34	0.27	0.23	0.18
Aggressive											
fetches	11974	12937	15104	16457	17588	18048	17824	17547	16917	16542	16254
driver time (sec)	5.987	6.4685	7.552	8.2285	8.794	9.024	8.912	8.7735	8.4585	8.271	8.127
stall time (sec)	18.727	4.342	1.132	0.107	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	98.815	84.911	82.785	82.436	82.895	83.126	83.013	82.883	82.564	82.373	82.228
average fetch time (msec)	8.219	12.54	15.309	16.142	16.775	16.717	16.777	16.966	17.597	17.96	18.603
average disk utilization	1	0.96	0.93	0.81	0.71	0.6	0.51	0.45	0.36	0.3	0.23
Reverse Aggressive											
fetches	12228	12814	12501	12033	11880	11837	11852	11883	11919	11954	12004
driver time (sec)	6.114	6.407	6.2505	6.0165	5.94	5.9185	5.926	5.9415	5.9595	5.977	6.002
stall time (sec)	27.37	6.138	1.178	0.195	0.005	0.013	0.011	0.009	0.005	0.016	0.008
elapsed time (sec)	107.585	86.646	81.529	80.312	80.046	80.032	80.038	80.051	80.065	80.094	80.111
average fetch time (msec)	8.408	12.53	15.639	16.608	16.992	17.386	17.832	18.199	19.542	18.83	19.076
average disk utilization	0.96	0.93	0.8	0.62	0.5	0.43	0.38	0.34	0.28	0.23	0.18

Table A.15: Performance on the glimpse trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493
driver time (sec)	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465
stall time (sec)	69.337	31.846	19.994	12.419	8.631	6.095	4.014	3.171	1.547	0.745	0.388
elapsed time (sec)	111.3	73.809	61.957	54.382	50.594	48.058	45.977	45.134	43.51	42.708	42.351
average fetch time (msec)	14.011	15.68	16.63	17.531	18.305	18.515	18.611	18.648	18.529	18.586	18.675
average disk utilization	0.82	0.69	0.58	0.52	0.47	0.42	0.38	0.34	0.28	0.24	0.18
Aggressive											
fetches	6690	6786	7277	7400	8361	9196	10233	10959	11953	11500	11253
driver time (sec)	3.345	3.393	3.6385	3.7	4.1805	4.598	5.1165	5.4795	5.9765	5.75	5.6265
stall time (sec)	59.899	23.204	10.316	3.068	1.321	0.252	0	0.009	0.005	0.001	0
elapsed time (sec)	101.96	65.313	52.671	45.484	44.218	43.566	43.833	44.205	44.698	44.467	44.343
average fetch time (msec)	13.814	15.675	16.321	17.255	17.756	17.404	17.228	17.12	16.022	16.038	16.341
average disk utilization	0.91	0.81	0.75	0.7	0.67	0.61	0.57	0.53	0.43	0.35	0.26
Reverse Aggressive											
fetches	6712	7179	7630	8141	7619	6803	6656	6709	6750	6822	6978
driver time (sec)	3.356	3.5895	3.815	4.0705	3.8095	3.4015	3.328	3.3545	3.375	3.411	3.489
stall time (sec)	59.04	22.734	8.948	1.776	1.094	0	0.011	0.009	0.005	0.006	0
elapsed time (sec)	101.112	65.04	51.479	44.563	43.62	42.118	42.055	42.08	42.096	42.133	42.205
average fetch time (msec)	14.018	15.73	16.272	16.806	18.093	18.293	18.44	18.529	18.633	18.563	18.547
average disk utilization	0.93	0.87	0.8	0.77	0.63	0.49	0.42	0.37	0.3	0.25	0.19

Table A.16: Performance on the ld trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	2904	2904	2904	2904	2904	2904	2904	2904	2904	2904	2904
driver time (sec)	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452	1.452
stall time (sec)	16.029	7.491	4.39	2.843	1.865	1.291	0.913	0.839	0.431	0.218	0.176
elapsed time (sec)	25.646	17.108	14.007	12.46	11.482	10.908	10.53	10.456	10.048	9.835	9.793
average fetch time (msec)	8.63	11.143	13.432	15.137	16.323	17.033	17.311	17.876	18.615	18.958	19.23
average disk utilization	0.98	0.95	0.93	0.88	0.83	0.76	0.68	0.62	0.54	0.47	0.36
Aggressive											
fetches	2943	2979	3000	3052	3228	3459	3712	3809	4061	4285	4608
driver time (sec)	1.4715	1.4895	1.5	1.526	1.614	1.7295	1.856	1.9045	2.0305	2.1425	2.304
stall time (sec)	15.651	7.305	3.955	2.126	0.983	0.509	0.017	0.009	0.005	0.001	0
elapsed time (sec)	25.287	16.959	13.62	11.817	10.762	10.403	10.038	10.078	10.2	10.308	10.469
average fetch time (msec)	8.575	11.002	13.144	14.781	15.716	16.17	16.718	16.973	17.723	18.074	18.23
average disk utilization	1	0.97	0.97	0.95	0.94	0.9	0.88	0.8	0.71	0.63	0.5
Reverse Aggressive											
fetches	3041	3079	3202	3312	3161	3037	3103	3000	2953	3004	3008
driver time (sec)	1.5205	1.5395	1.601	1.656	1.5805	1.5185	1.5515	1.5	1.4765	1.502	1.504
stall time (sec)	15.977	7.36	4.121	2.409	1.197	0.82	0.412	0.2	0.035	0.016	0.008
elapsed time (sec)	25.662	17.064	13.887	12.23	10.942	10.503	10.128	9.865	9.676	9.683	9.677
average fetch time (msec)	8.425	10.763	12.743	14.394	15.897	16.876	17.324	17.799	18.616	19.074	19.137
average disk utilization	1	0.97	0.98	0.97	0.92	0.81	0.76	0.68	0.57	0.49	0.37

Table A.17: Performance on the postgres-join trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	3856	3856	3856	3856	3856	3856
driver time (sec)	1.928	1.928	1.928	1.928	1.928	1.928
stall time (sec)	8.916	0.017	0.017	0.017	0.017	0.017
elapsed time (sec)	90.06	81.161	81.161	81.161	81.161	81.161
average fetch time (msec)	18.516	18.188	18.122	18.342	18.369	18.109
average disk utilization	0.79	0.43	0.29	0.22	0.17	0.14
Aggressive						
fetches	4138	5704	6188	6156	5978	5949
driver time (sec)	2.069	2.852	3.094	3.078	2.989	2.9745
stall time (sec)	8.546	0	0	0	0	0.001
elapsed time (sec)	89.831	82.068	82.31	82.294	82.205	82.191
average fetch time (msec)	18.584	17.688	16.933	17.105	17.293	17.357
average disk utilization	0.86	0.61	0.42	0.32	0.25	0.21
Reverse Aggressive						
fetches	3987	3853	3859	3873	3879	3892
driver time (sec)	1.9935	1.9265	1.9295	1.9365	1.9395	1.946
stall time (sec)	8.427	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	89.636	81.163	81.164	81.169	81.17	81.175
average fetch time (msec)	18.51	18.173	18.06	18.344	18.335	18.037
average disk utilization	0.82	0.43	0.29	0.22	0.18	0.14

Table A.18: Performance on the postgres-select trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	39.131	13.893	6.434	3.239	1.313	0.489	0.186	0.018	0.028	0.018	0.018
elapsed time (sec)	52.151	26.913	19.454	16.259	14.333	13.509	13.206	13.038	13.048	13.038	13.038
average fetch time (msec)	16.703	16.165	16.017	15.85	15.708	15.656	15.632	15.526	15.495	15.414	15.117
average disk utilization	0.99	0.93	0.85	0.75	0.68	0.6	0.52	0.46	0.37	0.3	0.22
Aggressive											
fetches	3085	3085	3085	3085	3085	3234	3707	3891	3926	3882	3731
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.617	1.8535	1.9455	1.963	1.941	1.8655
stall time (sec)	39.072	13.453	4.9	1.342	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	52.092	26.473	17.92	14.362	13.02	13.096	13.331	13.432	13.446	13.42	13.343
average fetch time (msec)	16.703	16.214	16.06	15.91	15.742	15.512	15.774	15.251	14.928	14.962	15.202
average disk utilization	0.99	0.94	0.92	0.85	0.75	0.64	0.63	0.55	0.44	0.36	0.27
Reverse Aggressive											
fetches	3106	3106	3318	3110	3109	3108	3112	3122	3116	3122	3124
driver time (sec)	1.553	1.553	1.659	1.555	1.5545	1.554	1.556	1.561	1.558	1.561	1.562
stall time (sec)	39.031	13.202	5.806	1.008	0	0.001	0	0	0	0	0.002
elapsed time (sec)	52.062	26.233	18.943	14.041	13.032	13.033	13.034	13.039	13.036	13.039	13.042
average fetch time (msec)	16.628	16.156	16.057	15.859	15.694	15.667	15.619	15.497	15.469	15.426	15.262
average disk utilization	0.99	0.96	0.94	0.88	0.75	0.62	0.53	0.46	0.37	0.31	0.23

Table A.19: Performance on the synth trace.

Disks	1	2	3	4
Fixed Horizon				
fetches	38000	38000	38000	38000
driver time (sec)	19	19	19	19
stall time (sec)	82.583	12.044	0	0
elapsed time (sec)	201.439	130.9	118.856	118.856
average fetch time (msec)	3.748	3.776	3.229	3.214
average disk utilization	0.71	0.55	0.34	0.26
Aggressive				
fetches	39240	41902	100994	100548
driver time (sec)	19.62	20.951	50.497	50.274
stall time (sec)	36.37	0.933	0.015	0.015
elapsed time (sec)	155.846	121.74	150.368	150.145
average fetch time (msec)	3.965	5.647	3.37	3.164
average disk utilization	1	0.97	0.75	0.53
Reverse Aggressive				
fetches	39265	42000	37907	38148
driver time (sec)	19.6325	21	18.9535	19.074
stall time (sec)	41.599	2.765	0.014	0.015
elapsed time (sec)	161.088	123.621	118.824	118.945
average fetch time (msec)	3.928	3.907	3.762	3.958
average disk utilization	0.96	0.66	0.4	0.32

Table A.20: Performance on the xds trace.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	5883	5883	5883	5883	5883	5883
driver time (sec)	2.9415	2.9415	2.9415	2.9415	2.9415	2.9415
stall time (sec)	34.937	8.068	4.974	2.068	1.005	0.242
elapsed time (sec)	68.644	41.775	38.681	35.775	34.712	33.949
average fetch time (msec)	10.86	7.769	14.127	10.142	14.97	10.835
average disk utilization	0.93	0.55	0.72	0.42	0.51	0.31
Aggressive						
fetches	5925	7662	6439	9847	8206	10114
driver time (sec)	2.9625	3.831	3.2195	4.9235	4.103	5.057
stall time (sec)	31.431	0.012	0	0	0	0
elapsed time (sec)	64.472	33.922	33.298	35.002	34.182	35.136
average fetch time (msec)	10.846	7.59	14.215	9.807	15.443	10.636
average disk utilization	1	0.86	0.92	0.69	0.74	0.51
Reverse Aggressive						
fetches	5910	5997	5945	6007	5904	6024
driver time (sec)	2.955	2.9985	2.9725	3.0035	2.952	3.012
stall time (sec)	31.298	0.043	0.334	0.01	0.012	0.011
elapsed time (sec)	65.018	33.807	34.072	33.779	33.729	33.788
average fetch time (msec)	10.913	7.729	14.213	9.877	15.115	10.124
average disk utilization	0.99	0.69	0.83	0.44	0.53	0.3

A.3 Performance data: double-speed CPU

This section contains the data for the xds trace, with the processor speed doubled.

Table A.21: Performance on the xds trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fixed Horizon											
fetches	5900	5900	5900	5900	5900	5900	5883	5883	5883	5883	5883
driver time (sec)	1.475	1.475	1.475	1.475	1.475	1.475	1.47075	1.47075	1.47075	1.47075	1.47075
stall time (sec)	47.186	16.135	13.901	7.272	6.058	2.778	5.577	5.343	3.038	3.306	2.317
elapsed time (sec)	63.698	32.647	30.413	23.784	22.57	19.29	22.422	22.188	19.883	20.151	19.162
average fetch time (msec)	10.714	7.735	14.014	10.122	15.41	10.959	15.97	12.872	12.847	14.048	13.583
average disk utilization	0.99	0.7	0.91	0.63	0.81	0.56	0.6	0.43	0.38	0.34	0.26
Aggressive											
fetches	5890	6272	5965	6471	5963	7602	6567	8701	10278	10607	10948
driver time (sec)	1.4725	1.568	1.49125	1.61775	1.49075	1.9005	1.64175	2.17525	2.5695	2.65175	2.737
stall time (sec)	46.755	10.572	11.808	2.384	2.776	0.048	0.097	0.045	0.046	0.036	0.026
elapsed time (sec)	63.264	27.177	28.336	19.038	19.303	16.985	17.113	17.595	17.99	18.062	18.137
average fetch time (msec)	10.737	7.664	14.121	10.064	15.496	10.871	16.398	12.921	12.7	13.709	12.392
average disk utilization	1	0.88	0.99	0.86	0.96	0.81	0.9	0.8	0.73	0.67	0.47
Reverse Aggressive											
fetches	5892	6095	5939	6182	6001	6017	5970	6042	6055	6090	6164
driver time (sec)	1.473	1.52375	1.48475	1.5455	1.50025	1.50425	1.4925	1.5105	1.51375	1.5225	1.541
stall time (sec)	47.078	8.592	11.946	1.135	2.836	0.055	0.304	0.045	0.046	0.036	0.034
elapsed time (sec)	63.588	25.152	28.467	17.717	19.373	16.596	17.171	16.93	16.934	16.933	16.949
average fetch time (msec)	10.787	7.734	14.132	10.228	15.506	10.787	16.032	12.257	11.954	12.902	12.794
average disk utilization	1	0.94	0.98	0.89	0.96	0.65	0.8	0.55	0.43	0.39	0.29

A.4 Performance data: varying cache size

This section contains the data for several traces with cache sizes of 5MB (640 blocks) and 15 MB (1920 blocks).

Table A.22: Performance on the glimpse trace, cache size 640.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	7804	7804	7804	7804	7804	7804
driver time (sec)	3.902	3.902	3.902	3.902	3.902	3.902
stall time (sec)	80.295	38.262	24.527	16.624	12.193	9.234
elapsed time (sec)	122.913	80.88	67.145	59.242	54.811	51.852
average fetch time (msec)	13.65	15.451	16.454	17.39	18.142	18.402
average disk utilization	0.87	0.75	0.64	0.57	0.52	0.46
Aggressive						
fetches	8133	8407	9377	8777	9847	10220
driver time (sec)	4.0665	4.2035	4.6885	4.3885	4.9235	5.11
stall time (sec)	73.144	27.576	12.83	4.352	0.873	0.007
elapsed time (sec)	115.927	70.496	56.235	47.457	44.513	43.833
average fetch time (msec)	13.02	14.281	14.479	16.104	15.938	16.928
average disk utilization	0.91	0.85	0.8	0.74	0.71	0.66
Reverse Aggressive						
fetches	8280	9007	9967	9318	9802	8251
driver time (sec)	4.14	4.5035	4.9835	4.659	4.901	4.1255
stall time (sec)	69.892	24.098	7.214	1.796	0.064	0
elapsed time (sec)	112.748	67.318	50.914	45.171	43.681	42.842
average fetch time (msec)	12.649	13.017	13.017	13.804	14.166	17.142
average disk utilization	0.93	0.87	0.85	0.71	0.64	0.55

Table A.23: Performance on the glimpse trace, cache size 1920.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	5853	5853	5853	5853	5853	5853
driver time (sec)	2.9265	2.9265	2.9265	2.9265	2.9265	2.9265
stall time (sec)	58.697	27.363	18.29	12.82	9.592	7.442
elapsed time (sec)	100.34	69.006	59.933	54.463	51.235	49.085
average fetch time (msec)	13.302	14.879	16.005	17.096	17.981	18.251
average disk utilization	0.78	0.63	0.52	0.46	0.41	0.36
Aggressive						
fetches	6041	6121	6647	7044	8048	8390
driver time (sec)	3.0205	3.0605	3.3235	3.522	4.024	4.195
stall time (sec)	46.429	13.448	4.781	2.51	0.829	0.029
elapsed time (sec)	88.166	55.225	46.821	44.748	43.569	42.94
average fetch time (msec)	12.832	14.085	14.453	16.047	16.247	17.082
average disk utilization	0.88	0.78	0.68	0.63	0.6	0.56
Reverse Aggressive						
fetches	5998	6072	6542	6377	6441	6035
driver time (sec)	2.999	3.036	3.271	3.1885	3.2205	3.0175
stall time (sec)	42.301	11.025	3.324	0.964	0	0
elapsed time (sec)	84.016	52.777	45.311	42.869	41.937	41.734
average fetch time (msec)	12.749	13.675	13.685	14.398	14.649	17.02
average disk utilization	0.91	0.79	0.66	0.54	0.45	0.41

Table A.24: Performance on the postgres-join trace, cache size 640.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	4640	4640	4640	4640	4640	4640
driver time (sec)	2.32	2.32	2.32	2.32	2.32	2.32
stall time (sec)	4.96	0.04	0.017	0.017	0.017	0.017
elapsed time (sec)	86.496	81.576	81.553	81.553	81.553	81.553
average fetch time (msec)	17.211	18.089	18.152	18.378	18.463	18.217
average disk utilization	0.92	0.51	0.34	0.26	0.21	0.17
Aggressive						
fetches	5378	7310	8013	8043	7758	7561
driver time (sec)	2.689	3.655	4.0065	4.0215	3.879	3.7805
stall time (sec)	3.994	0.152	0.258	0	0	0.001
elapsed time (sec)	85.899	83.023	83.48	83.237	83.095	82.997
average fetch time (msec)	15.152	17.332	16.677	16.709	16.899	16.909
average disk utilization	0.95	0.76	0.53	0.4	0.32	0.26
Reverse Aggressive						
fetches	4912	4615	4631	4657	4676	4691
driver time (sec)	2.456	2.3075	2.3155	2.3285	2.338	2.3455
stall time (sec)	3.655	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	85.327	81.544	81.55	81.561	81.569	81.574
average fetch time (msec)	16.064	18.147	18.068	18.361	18.282	18.026
average disk utilization	0.92	0.51	0.34	0.26	0.21	0.17

Table A.25: Performance on the postgres-join trace, cache size 1920.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	3793	3793	3793	3793	3793	3793
driver time (sec)	1.8965	1.8965	1.8965	1.8965	1.8965	1.8965
stall time (sec)	4.723	0.041	0.018	0.018	0.018	0.018
elapsed time (sec)	85.835	81.153	81.13	81.13	81.13	81.13
average fetch time (msec)	17.261	18.038	18.066	18.353	18.357	18.126
average disk utilization	0.76	0.42	0.28	0.21	0.17	0.14
Aggressive						
fetches	3943	4797	4976	4943	4863	4856
driver time (sec)	1.9715	2.3985	2.488	2.4715	2.4315	2.428
stall time (sec)	3.995	0.153	0.258	0	0	0.001
elapsed time (sec)	85.182	81.767	81.962	81.687	81.647	81.645
average fetch time (msec)	14.781	16.013	15.626	16.613	16.646	17.033
average disk utilization	0.68	0.47	0.32	0.25	0.2	0.17
Reverse Aggressive						
fetches	3801	3795	3801	3801	3801	3801
driver time (sec)	1.9005	1.8975	1.9005	1.9005	1.9005	1.9005
stall time (sec)	3.775	0.009	0.001	0	0	0.001
elapsed time (sec)	84.891	81.122	81.117	81.116	81.116	81.117
average fetch time (msec)	14.786	17.122	17.184	17.237	17.173	17.464
average disk utilization	0.66	0.4	0.27	0.2	0.16	0.14

Table A.26: Performance on the postgres-select trace, cache size 640.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	3155	3155	3155	3155	3155	3155
driver time (sec)	1.5775	1.5775	1.5775	1.5775	1.5775	1.5775
stall time (sec)	32.755	12.841	6.049	3.226	1.402	0.581
elapsed time (sec)	45.81	25.896	19.104	16.281	14.457	13.636
average fetch time (msec)	14.222	14.786	14.943	15.012	15.279	15.323
average disk utilization	0.98	0.9	0.82	0.73	0.67	0.59
Aggressive						
fetches	3249	3299	3394	3317	3965	4099
driver time (sec)	1.6245	1.6495	1.697	1.6585	1.9825	2.0495
stall time (sec)	32.613	11.717	4.384	1.007	0	0.001
elapsed time (sec)	45.715	24.844	17.559	14.143	13.46	13.528
average fetch time (msec)	13.938	14.118	13.944	14.557	13.923	15.171
average disk utilization	0.99	0.94	0.9	0.85	0.82	0.77
Reverse Aggressive						
fetches	3274	3354	3290	3341	3191	3170
driver time (sec)	1.637	1.677	1.645	1.6705	1.5955	1.585
stall time (sec)	31.026	10.967	4.271	0.353	0.005	0.013
elapsed time (sec)	44.141	24.122	17.394	13.501	13.078	13.076
average fetch time (msec)	13.282	13.614	14.029	13.744	15.048	15.574
average disk utilization	0.99	0.95	0.88	0.85	0.73	0.63

Table A.27: Performance on the postgres-select trace, cache size 1920.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	32.37	12.647	5.943	3.154	1.402	0.581
elapsed time (sec)	45.39	25.667	18.963	16.174	14.422	13.601
average fetch time (msec)	14.368	14.906	15.044	15.13	15.347	15.413
average disk utilization	0.98	0.9	0.82	0.72	0.66	0.58
Aggressive						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.691	10.772	3.517	0.844	0	0.001
elapsed time (sec)	43.711	23.792	16.537	13.864	13.02	13.021
average fetch time (msec)	13.985	14.173	13.95	14.55	14.446	15.175
average disk utilization	0.99	0.92	0.87	0.81	0.68	0.6
Reverse Aggressive						
fetches	3106	3106	3122	3110	3109	3108
driver time (sec)	1.553	1.553	1.561	1.555	1.5545	1.554
stall time (sec)	28.956	8.461	2.656	0.125	0	0.001
elapsed time (sec)	41.987	21.492	15.695	13.158	13.032	13.033
average fetch time (msec)	13.248	12.668	12.072	13.581	14.419	15.115
average disk utilization	0.98	0.92	0.8	0.8	0.69	0.6

Table A.28: Performance on the xds trace, cache size 640.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	6726	6726	6726	6726	6726	6726
driver time (sec)	3.363	3.363	3.363	3.363	3.363	3.363
stall time (sec)	38.883	6.666	4.75	1.751	0.653	0.128
elapsed time (sec)	72.325	40.108	38.192	35.193	34.095	33.57
average fetch time (msec)	10.533	7.597	14.077	9.972	15.262	10.713
average disk utilization	0.98	0.64	0.83	0.48	0.6	0.36
Aggressive						
fetches	6866	8769	7501	10876	8913	11309
driver time (sec)	3.433	4.3845	3.7505	5.438	4.4565	5.6545
stall time (sec)	38.711	1.996	1.719	0.129	0.134	0.055
elapsed time (sec)	72.223	36.459	35.548	35.646	34.669	35.788
average fetch time (msec)	10.484	7.313	13.856	9.646	15.229	10.67
average disk utilization	1	0.88	0.97	0.74	0.78	0.56
Reverse Aggressive						
fetches	6718	6983	6813	7070	6712	7040
driver time (sec)	3.359	3.4915	3.4065	3.535	3.356	3.52
stall time (sec)	37.569	0.371	1.292	0.096	0.016	0.023
elapsed time (sec)	71.007	33.941	34.777	33.71	33.451	33.622
average fetch time (msec)	10.527	7.58	14.184	9.639	14.91	10.011
average disk utilization	1	0.78	0.93	0.51	0.6	0.35

Table A.29: Performance on the xds trace, cache size 1920.

Disks	1	2	3	4	5	6
Fixed Horizon						
fetches	5392	5392	5392	5392	5392	5392
driver time (sec)	2.696	2.696	2.696	2.696	2.696	2.696
stall time (sec)	29.047	4.104	2.404	0.892	0.304	0.094
elapsed time (sec)	61.822	36.879	35.179	33.667	33.079	32.869
average fetch time (msec)	10.857	7.848	13.977	10.251	14.806	10.846
average disk utilization	0.95	0.57	0.71	0.41	0.48	0.3
Aggressive						
fetches	5392	7067	5894	8817	7483	9174
driver time (sec)	2.696	3.5335	2.947	4.4085	3.7415	4.587
stall time (sec)	26.626	0.337	0.355	0.129	0.134	0.055
elapsed time (sec)	59.401	33.949	33.381	34.616	33.954	34.721
average fetch time (msec)	10.841	7.613	14.085	10.061	15.434	10.683
average disk utilization	0.98	0.79	0.83	0.64	0.68	0.47
Reverse Aggressive						
fetches	5415	5531	5415	5530	5396	5538
driver time (sec)	2.7075	2.7655	2.7075	2.765	2.698	2.769
stall time (sec)	26.357	0.107	0.206	0.02	0.016	0.017
elapsed time (sec)	59.143	32.951	32.992	32.864	32.793	32.865
average fetch time (msec)	10.828	7.809	14.095	9.927	14.775	10.376
average disk utilization	0.99	0.66	0.77	0.42	0.49	0.29

A.5 Performance data: varying *aggressive*'s batch size

This section contains the performance data for *aggressive* with varying batch size.

Table A.30: Aggressive performance as a function of batch size on the dinero trace.

Disks	1	2	3	4	5	6
Batch size 4						
fetches	8812	8813	8812	8812	8812	8813
driver time (sec)	4.406	4.4065	4.406	4.406	4.406	4.4065
stall time (sec)	0.023	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	107.967	107.966	107.963	107.961	107.959	107.958
average fetch time (msec)	3.118	3.134	3.151	3.167	3.18	3.193
average disk utilization	0.25	0.13	0.086	0.065	0.052	0.043
Batch size 8						
fetches	8812	8812	8812	8813	8817	8816
driver time (sec)	4.406	4.406	4.406	4.4065	4.4085	4.408
stall time (sec)	0.021	0.017	0.013	0.009	0.005	0.001
elapsed time (sec)	107.965	107.961	107.957	107.954	107.952	107.947
average fetch time (msec)	3.118	3.135	3.155	3.174	3.182	3.203
average disk utilization	0.25	0.13	0.086	0.065	0.052	0.044
Batch size 16						
fetches	8812	8812	8812	8815	8812	8838
driver time (sec)	4.406	4.406	4.406	4.4075	4.406	4.419
stall time (sec)	0.017	0.009	0.001	0	0	0
elapsed time (sec)	107.961	107.953	107.945	107.946	107.944	107.957
average fetch time (msec)	3.131	3.14	3.162	3.176	3.188	3.206
average disk utilization	0.26	0.13	0.086	0.065	0.052	0.044
Batch size 40						
fetches	8812	8812	8823	8852	8869	8826
driver time (sec)	4.406	4.406	4.4115	4.426	4.4345	4.413
stall time (sec)	0.015	0	0	0	0	0
elapsed time (sec)	107.959	107.944	107.95	107.964	107.973	107.951
average fetch time (msec)	3.141	3.146	3.174	3.198	3.217	3.229
average disk utilization	0.26	0.13	0.086	0.066	0.053	0.044
Batch size 80						
fetches	8812	8813	8812	8853	8883	8812
driver time (sec)	4.406	4.4065	4.406	4.4265	4.4415	4.406
stall time (sec)	0.145	0	0.015	0	0	0
elapsed time (sec)	108.089	107.945	107.959	107.965	107.98	107.944
average fetch time (msec)	3.141	3.148	3.185	3.206	3.221	3.23
average disk utilization	0.26	0.13	0.087	0.066	0.053	0.044
Batch size 160						
fetches	8812	8814	8840	8812	8812	8812
driver time (sec)	4.406	4.407	4.42	4.406	4.406	4.406
stall time (sec)	0.405	0	0.195	0.055	0	0
elapsed time (sec)	108.349	107.945	108.153	107.999	107.944	107.944
average fetch time (msec)	3.15	3.163	3.19	3.206	3.215	3.231
average disk utilization	0.26	0.13	0.087	0.065	0.052	0.044

Table A.31: Aggressive performance as a function of batch size on the cscope1 trace.

Disks	1	2	3	4	5	6
Batch size 4						
fetches	5325	8555	8599	8661	8621	8583
driver time (sec)	2.6625	4.2775	4.2995	4.3305	4.3105	4.2915
stall time (sec)	4.161	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	31.758	29.233	29.253	29.282	29.26	29.239
average fetch time (msec)	5.28	3.349	3.374	3.383	3.366	3.376
average disk utilization	0.89	0.49	0.33	0.25	0.2	0.17
Batch size 8						
fetches	5802	8584	8624	8650	8635	8576
driver time (sec)	2.901	4.292	4.312	4.325	4.3175	4.288
stall time (sec)	1.812	0.017	0.013	0.009	0.005	0.001
elapsed time (sec)	29.647	29.243	29.259	29.268	29.257	29.223
average fetch time (msec)	4.513	3.373	3.379	3.367	3.37	3.356
average disk utilization	0.88	0.5	0.33	0.25	0.2	0.16
Batch size 16						
fetches	6300	8585	8647	8678	8621	8572
driver time (sec)	3.15	4.2925	4.3235	4.339	4.3105	4.286
stall time (sec)	0.881	0.009	0.001	0	0	0
elapsed time (sec)	28.965	29.236	29.259	29.273	29.245	29.22
average fetch time (msec)	3.991	3.416	3.41	3.365	3.39	3.38
average disk utilization	0.87	0.5	0.34	0.25	0.2	0.17
Batch size 40						
fetches	6616	8570	8672	8662	8705	8574
driver time (sec)	3.308	4.285	4.336	4.331	4.3525	4.287
stall time (sec)	0.665	0	0	0	0	0
elapsed time (sec)	28.907	29.219	29.27	29.265	29.287	29.221
average fetch time (msec)	3.796	3.361	3.429	3.381	3.421	3.389
average disk utilization	0.87	0.49	0.34	0.25	0.2	0.17
Batch size 80						
fetches	6931	8570	8677	8747	8713	8572
driver time (sec)	3.4655	4.285	4.3385	4.3735	4.3565	4.286
stall time (sec)	0.911	0	0.007	0	0	0
elapsed time (sec)	29.311	29.219	29.28	29.308	29.291	29.22
average fetch time (msec)	3.758	3.374	3.465	3.423	3.424	3.404
average disk utilization	0.89	0.49	0.34	0.26	0.2	0.17
Batch size 160						
fetches	7360	8699	8662	8678	8621	8572
driver time (sec)	3.68	4.3495	4.331	4.339	4.3105	4.286
stall time (sec)	1.83	0.192	0.187	0	0	0
elapsed time (sec)	30.444	29.476	29.452	29.273	29.245	29.22
average fetch time (msec)	3.757	3.436	3.417	3.413	3.412	3.407
average disk utilization	0.91	0.51	0.33	0.25	0.2	0.17

Table A.32: Aggressive performance as a function of batch size on the cscope2 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Batch size 4											
fetches	5982	6098	7029	8240	9574	10678	11406	11717	11619	11102	10662
driver time (sec)	2.991	3.049	3.5145	4.12	4.787	5.339	5.703	5.8585	5.8095	5.551	5.331
stall time (sec)	24.775	9.407	3.736	0.807	0.015	0.013	0.011	0.009	0.005	0.001	0
elapsed time (sec)	64.875	49.565	44.36	42.036	41.911	42.461	42.823	42.977	42.924	42.661	42.44
average fetch time (msec)	10.407	15.336	16.97	17.65	17.787	17.891	17.752	17.551	17.224	17.65	18.201
average disk utilization	0.96	0.94	0.9	0.86	0.81	0.75	0.68	0.6	0.47	0.38	0.29
Batch size 8											
fetches	6009	6194	7277	8541	9952	11014	11587	11758	11565	11134	10662
driver time (sec)	3.0045	3.097	3.6385	4.2705	4.976	5.507	5.7935	5.879	5.7825	5.567	5.331
stall time (sec)	23.367	8.442	3.15	0.299	0.005	0.001	0	0	0	0	0
elapsed time (sec)	63.481	48.648	43.898	41.679	42.09	42.617	42.903	42.988	42.892	42.676	42.44
average fetch time (msec)	10.298	14.97	16.392	17.193	17.527	17.683	17.722	17.532	17.195	17.534	18.135
average disk utilization	0.97	0.95	0.91	0.88	0.83	0.76	0.68	0.6	0.46	0.38	0.28
Batch size 16											
fetches	6044	6321	7583	8956	10299	11364	11617	11758	11565	11134	10662
driver time (sec)	3.022	3.1605	3.7915	4.478	5.1495	5.682	5.8085	5.879	5.7825	5.567	5.331
stall time (sec)	20.041	6.887	2.507	0	0	0	0	0	0	0	0
elapsed time (sec)	60.172	47.157	43.408	41.587	42.259	42.791	42.918	42.988	42.892	42.676	42.44
average fetch time (msec)	9.793	14.201	15.654	16.514	17.138	17.426	17.444	17.343	17.098	17.498	18.085
average disk utilization	0.98	0.95	0.91	0.89	0.84	0.77	0.67	0.59	0.46	0.38	0.28
Batch size 40											
fetches	6171	6592	8208	9684	10892	11553	11728	11884	11654	11164	10662
driver time (sec)	3.0855	3.296	4.104	4.842	5.446	5.7765	5.864	5.942	5.827	5.582	5.331
stall time (sec)	16.349	5.597	1.798	0.156	0.044	0.115	0.06	0.046	0	0	0
elapsed time (sec)	56.544	46.002	43.011	42.107	42.599	43.001	43.033	43.097	42.936	42.691	42.44
average fetch time (msec)	9.099	13.256	14.354	15.55	16.49	17.067	17.266	17.228	16.938	17.369	17.931
average disk utilization	0.99	0.95	0.91	0.89	0.84	0.76	0.67	0.59	0.46	0.38	0.28
Batch size 80											
fetches	6318	7022	8799	10463	11331	11655	11753	11897	11807	11291	10662
driver time (sec)	3.159	3.511	4.3995	5.2315	5.6655	5.8275	5.8765	5.9485	5.9035	5.6455	5.331
stall time (sec)	15.858	4.803	1.017	0.48	0.285	0.4	0.197	0.183	0.091	0.115	0
elapsed time (sec)	56.126	45.423	42.526	42.821	43.06	43.337	43.183	43.241	43.104	42.87	42.44
average fetch time (msec)	8.773	12.278	13.331	14.736	16.237	16.979	17.106	17.009	16.756	17.182	17.717
average disk utilization	0.99	0.95	0.92	0.9	0.85	0.76	0.67	0.58	0.46	0.38	0.28
Batch size 160											
fetches	6771	7778	9478	10967	11325	11942	11859	12039	11619	11130	10604
driver time (sec)	3.3855	3.889	4.739	5.4835	5.6625	5.971	5.9295	6.0195	5.8095	5.565	5.302
stall time (sec)	17.081	5.856	1.078	1.119	1.029	0.956	0.832	0.882	0.559	0.197	0.016
elapsed time (sec)	57.576	46.854	42.926	43.712	43.801	44.036	43.871	44.011	43.478	42.871	42.427
average fetch time (msec)	8.443	11.569	12.688	14.445	16.183	16.833	17.135	17.078	16.679	17.019	17.744
average disk utilization	0.99	0.96	0.93	0.91	0.84	0.76	0.66	0.58	0.45	0.37	0.28

Table A.33: Aggressive performance as a function of batch size on the cscope3 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Batch size 4											
fetches	11763	12349	14291	16061	17462	18026	17821	17577	16917	16542	16314
driver time (sec)	5.8815	6.1745	7.1455	8.0305	8.731	9.013	8.9105	8.7885	8.4585	8.271	8.157
stall time (sec)	28.053	6.26	1.518	0.363	0.015	0.013	0.011	0.009	0.005	0.001	0
elapsed time (sec)	108.035	86.535	82.764	82.494	82.847	83.127	83.022	82.898	82.564	82.373	82.258
average fetch time (msec)	8.624	12.843	15.411	16.314	16.698	16.667	16.725	16.905	17.605	17.966	18.49
average disk utilization	0.94	0.92	0.89	0.79	0.7	0.6	0.51	0.45	0.36	0.3	0.23
Batch size 8											
fetches	11779	12627	14881	16441	17635	18081	17894	17565	16902	16546	16284
driver time (sec)	5.8895	6.3135	7.4405	8.2205	8.8175	9.0405	8.947	8.7825	8.451	8.273	8.142
stall time (sec)	25.287	5.205	1.323	0.069	0.005	0.001	0	0	0	0	0
elapsed time (sec)	105.277	85.619	82.864	82.39	82.923	83.142	83.048	82.883	82.552	82.374	82.243
average fetch time (msec)	8.623	12.65	15.121	16.158	16.662	16.568	16.711	16.84	17.523	17.942	18.445
average disk utilization	0.96	0.93	0.91	0.81	0.71	0.6	0.51	0.45	0.36	0.3	0.23
Batch size 16											
fetches	11811	13043	15366	16740	17713	18175	17924	17607	16902	16598	16314
driver time (sec)	5.9055	6.5215	7.683	8.37	8.8565	9.0875	8.962	8.8035	8.451	8.299	8.157
stall time (sec)	21.462	3.911	1.079	0.052	0	0.036	0	0	0	0.033	0
elapsed time (sec)	101.468	84.533	82.863	82.523	82.957	83.224	83.063	82.904	82.552	82.433	82.258
average fetch time (msec)	8.416	12.214	14.719	15.92	16.553	16.541	16.552	16.811	17.462	17.882	18.404
average disk utilization	0.98	0.94	0.91	0.81	0.71	0.6	0.51	0.45	0.36	0.3	0.23
Batch size 40											
fetches	11925	13572	15938	17104	17842	18158	17924	17748	16981	16646	16344
driver time (sec)	5.9625	6.786	7.969	8.552	8.921	9.079	8.962	8.874	8.4905	8.323	8.172
stall time (sec)	15.648	2.862	0.64	0.274	0.264	0.249	0	0.06	0.179	0.204	0
elapsed time (sec)	95.711	83.749	82.71	82.927	83.286	83.429	83.063	83.035	82.77	82.628	82.273
average fetch time (msec)	7.949	11.597	14.215	15.737	16.3	16.35	16.496	16.651	17.391	17.777	18.264
average disk utilization	0.99	0.94	0.91	0.81	0.7	0.59	0.51	0.44	0.36	0.3	0.23
Batch size 80											
fetches	12092	14105	16543	17257	17919	18357	18137	17819	16963	16766	16275
driver time (sec)	6.046	7.0525	8.2715	8.6285	8.9595	9.1785	9.0685	8.9095	8.4815	8.383	8.1375
stall time (sec)	13.943	2.195	0.715	0.584	0.704	0.489	0.219	0.368	0.494	0.338	0.093
elapsed time (sec)	94.09	83.348	83.087	83.313	83.764	83.768	83.388	83.378	83.076	82.822	82.331
average fetch time (msec)	7.741	11.093	13.798	15.546	16.225	16.284	16.339	16.549	17.325	17.639	18.155
average disk utilization	0.99	0.94	0.92	0.81	0.69	0.59	0.51	0.44	0.35	0.3	0.22
Batch size 160											
fetches	12512	14919	16966	17314	18012	18450	18245	17733	16924	16468	16249
driver time (sec)	6.256	7.4595	8.483	8.657	9.006	9.225	9.1225	8.8665	8.462	8.234	8.1245
stall time (sec)	15.216	2.542	1.455	1.297	1.523	0.981	0.913	0.947	0.801	0.373	0.031
elapsed time (sec)	95.573	84.102	84.039	84.055	84.63	84.307	84.136	83.914	83.364	82.708	82.256
average fetch time (msec)	7.512	10.643	13.615	15.645	16.247	16.196	16.337	16.62	17.185	17.6	18.164
average disk utilization	0.98	0.94	0.92	0.81	0.69	0.59	0.51	0.44	0.35	0.29	0.22

Table A.34: Aggressive performance as a function of batch size on the glimpse trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Batch size 4											
fetches	6520	6597	6803	7178	7957	8946	10017	10992	12009	11530	11315
driver time (sec)	3.26	3.2985	3.4015	3.589	3.9785	4.473	5.0085	5.496	6.0045	5.765	5.6575
stall time (sec)	62.407	24.43	10.612	3.948	2.034	0.371	0.011	0.009	0.005	0.001	0
elapsed time (sec)	104.383	66.445	52.73	46.253	44.729	43.56	43.736	44.221	44.726	44.482	44.374
average fetch time (msec)	13.918	15.694	16.698	17.342	17.959	17.334	17.074	16.896	16.137	15.917	16.198
average disk utilization	0.87	0.78	0.72	0.67	0.64	0.59	0.56	0.52	0.43	0.34	0.26
Batch size 8											
fetches	6532	6636	6883	7349	8475	9376	10423	11296	12085	11709	11457
driver time (sec)	3.266	3.318	3.4415	3.6745	4.2375	4.688	5.2115	5.648	6.0425	5.8545	5.7285
stall time (sec)	60.737	22.944	9.255	3.264	1.495	0.035	0	0	0	0	0
elapsed time (sec)	102.719	64.978	51.413	45.655	44.449	43.439	43.928	44.364	44.759	44.571	44.445
average fetch time (msec)	13.732	15.366	16.344	16.857	16.92	16.836	16.79	16.886	15.76	15.862	16.169
average disk utilization	0.87	0.78	0.73	0.68	0.65	0.61	0.57	0.54	0.43	0.35	0.26
Batch size 16											
fetches	6553	6631	6959	7551	8908	9975	10860	11795	12487	11980	11499
driver time (sec)	3.2765	3.3155	3.4795	3.7755	4.454	4.9875	5.43	5.8975	6.2435	5.99	5.7495
stall time (sec)	59.057	20.883	7.529	2.495	0.826	0	0	0	0	0	0
elapsed time (sec)	101.05	62.915	49.725	44.987	43.996	43.704	44.146	44.614	44.96	44.706	44.466
average fetch time (msec)	13.523	15.002	15.658	16.247	15.973	16.255	16.513	16.529	15.635	15.695	16.172
average disk utilization	0.88	0.79	0.73	0.68	0.65	0.62	0.58	0.55	0.43	0.35	0.26
Batch size 40											
fetches	6601	6888	7287	8524	9998	10670	11662	12237	12721	12148	11517
driver time (sec)	3.3005	3.444	3.6435	4.262	4.999	5.335	5.831	6.1185	6.3605	6.074	5.7585
stall time (sec)	56.841	18.58	6.384	1.608	0	0	0	0	0	0	0
elapsed time (sec)	98.858	60.74	48.744	44.586	43.715	44.051	44.547	44.835	45.077	44.79	44.475
average fetch time (msec)	13.225	14.259	14.645	14.741	14.873	15.694	16.011	15.98	15.585	15.677	15.951
average disk utilization	0.88	0.81	0.73	0.7	0.68	0.63	0.6	0.55	0.44	0.35	0.26
Batch size 80											
fetches	6690	7128	7930	9430	11022	11778	12142	12540	12643	12026	11487
driver time (sec)	3.345	3.564	3.965	4.715	5.511	5.889	6.071	6.27	6.3215	6.013	5.7435
stall time (sec)	54.58	16.457	5.461	0.836	0	0.021	0	0.001	0	0	0
elapsed time (sec)	96.641	58.737	48.142	44.267	44.227	44.626	44.787	44.987	45.038	44.729	44.46
average fetch time (msec)	12.889	13.571	13.547	13.629	14.129	15.243	15.763	15.963	15.638	15.547	15.83
average disk utilization	0.89	0.82	0.74	0.73	0.7	0.67	0.61	0.56	0.44	0.35	0.26
Batch size 160											
fetches	7007	7690	9473	10657	11387	11825	12417	12521	12885	12062	11487
driver time (sec)	3.5035	3.845	4.7365	5.3285	5.6935	5.9125	6.2085	6.2605	6.4425	6.031	5.7435
stall time (sec)	51.598	17.187	5.473	1.041	0.7	0.575	0.27	0.37	0	0	0
elapsed time (sec)	93.818	59.748	48.926	45.086	45.11	45.204	45.195	45.347	45.159	44.747	44.46
average fetch time (msec)	12.508	12.9	12.754	13.102	14.031	15.132	15.654	15.918	15.514	15.515	15.792
average disk utilization	0.93	0.83	0.82	0.77	0.71	0.66	0.61	0.55	0.44	0.35	0.26

Table A.35: Aggressive performance as a function of batch size on the ld trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Batch size 4											
fetches	2885	2909	2916	2952	3048	3332	3586	3779	4091	4285	4651
driver time (sec)	1.4425	1.4545	1.458	1.476	1.524	1.666	1.793	1.8895	2.0455	2.1425	2.3255
stall time (sec)	16.476	8.221	5.117	2.955	1.414	0.737	0.174	0.009	0.005	0.001	0
elapsed time (sec)	26.083	17.84	14.74	12.596	11.103	10.568	10.132	10.063	10.215	10.308	10.49
average fetch time (msec)	8.679	11.449	13.866	15.474	16.32	16.472	17.13	17.175	17.62	18.017	18.261
average disk utilization	0.96	0.93	0.91	0.91	0.9	0.87	0.87	0.81	0.71	0.62	0.51
Batch size 8											
fetches	2892	2918	2942	3021	3192	3505	3734	3951	4183	4410	4687
driver time (sec)	1.446	1.459	1.471	1.5105	1.596	1.7525	1.867	1.9755	2.0915	2.205	2.3435
stall time (sec)	17.041	8.126	4.675	2.65	1.241	0.265	0.023	0.012	0	0	0
elapsed time (sec)	26.652	17.75	14.311	12.325	11.002	10.182	10.055	10.152	10.256	10.37	10.508
average fetch time (msec)	8.981	11.609	13.686	14.984	15.797	15.958	16.444	16.804	17.436	17.624	18.105
average disk utilization	0.97	0.95	0.94	0.92	0.92	0.92	0.87	0.82	0.71	0.62	0.5
Batch size 16											
fetches	2896	2942	2982	3102	3310	3626	3856	4107	4329	4559	4748
driver time (sec)	1.448	1.471	1.491	1.551	1.655	1.813	1.928	2.0535	2.1645	2.2795	2.374
stall time (sec)	16.552	7.34	3.845	2.052	0.579	0.287	0	0	0	0	0
elapsed time (sec)	26.165	16.976	13.501	11.768	10.399	10.265	10.093	10.218	10.329	10.444	10.539
average fetch time (msec)	8.919	11.194	12.99	14.199	14.932	15.516	15.914	16.277	17.002	17.214	17.786
average disk utilization	0.99	0.97	0.96	0.94	0.95	0.91	0.87	0.82	0.71	0.63	0.5
Batch size 40											
fetches	2934	2982	3137	3297	3560	3893	4105	4338	4610	4776	4741
driver time (sec)	1.467	1.491	1.5685	1.6485	1.78	1.9465	2.0525	2.169	2.305	2.388	2.3705
stall time (sec)	15.58	6.329	3.433	1.594	0.368	0.22	0.077	0.013	0	0	0
elapsed time (sec)	25.212	15.985	13.166	11.407	10.313	10.331	10.294	10.347	10.47	10.553	10.535
average fetch time (msec)	8.523	10.583	12.037	13.131	13.917	14.641	15.208	15.687	16.162	16.703	17.38
average disk utilization	0.99	0.99	0.96	0.95	0.96	0.92	0.87	0.82	0.71	0.63	0.49
Batch size 80											
fetches	2981	3142	3257	3571	3909	4326	4496	4692	4941	4862	4695
driver time (sec)	1.4905	1.571	1.6285	1.7855	1.9545	2.163	2.248	2.346	2.4705	2.431	2.3475
stall time (sec)	15.245	6.26	3.176	1.554	0.798	0.392	0.244	0.332	0.05	0.083	0
elapsed time (sec)	24.9	15.996	12.969	11.504	10.917	10.72	10.657	10.843	10.685	10.679	10.512
average fetch time (msec)	8.248	9.932	11.459	12.33	13.341	13.85	14.594	15.099	15.92	16.378	16.986
average disk utilization	0.99	0.98	0.96	0.96	0.96	0.93	0.88	0.82	0.74	0.62	0.47
Batch size 160											
fetches	3134	3356	3823	4207	4560	4644	4914	4721	4788	4688	4630
driver time (sec)	1.567	1.678	1.9115	2.1035	2.28	2.322	2.457	2.3605	2.394	2.344	2.315
stall time (sec)	15.141	6.183	3.941	2.35	1.72	1.466	0.879	0.953	0.441	0.386	0
elapsed time (sec)	24.873	16.026	14.017	12.618	12.165	11.953	11.501	11.478	11	10.895	10.48
average fetch time (msec)	7.856	9.362	10.522	11.344	12.389	13.653	14.519	15.356	15.746	16.049	17.023
average disk utilization	0.99	0.98	0.96	0.95	0.93	0.88	0.89	0.79	0.69	0.58	0.47

Table A.36: Aggressive performance as a function of batch size on the postgres-join trace.

Disks	1	2	3	4	5	6
Batch size 4						
fetches	3925	5451	6044	6062	5978	5925
driver time (sec)	1.9625	2.7255	3.022	3.031	2.989	2.9625
stall time (sec)	11.351	0.021	0.019	0.017	0.015	0.013
elapsed time (sec)	92.529	81.962	82.257	82.264	82.22	82.191
average fetch time (msec)	18.345	17.669	16.975	17.051	17.256	17.288
average disk utilization	0.78	0.59	0.42	0.31	0.25	0.21
Batch size 8						
fetches	4051	5642	6116	6078	5998	5919
driver time (sec)	2.0255	2.821	3.058	3.039	2.999	2.9595
stall time (sec)	9.318	0.017	0.013	0.009	0.005	0.001
elapsed time (sec)	90.559	82.054	82.287	82.264	82.22	82.176
average fetch time (msec)	17.772	17.483	16.701	16.848	17.034	17.102
average disk utilization	0.79	0.6	0.41	0.31	0.25	0.21
Batch size 16						
fetches	4233	5718	6170	6156	6047	5951
driver time (sec)	2.1165	2.859	3.085	3.078	3.0235	2.9755
stall time (sec)	7.437	0.009	0.001	0	0	0
elapsed time (sec)	88.769	82.084	82.302	82.294	82.239	82.191
average fetch time (msec)	16.928	17.15	16.438	16.578	16.706	16.837
average disk utilization	0.81	0.6	0.41	0.31	0.25	0.2
Batch size 40						
fetches	4510	5836	6225	6239	6137	5954
driver time (sec)	2.255	2.918	3.1125	3.1195	3.0685	2.977
stall time (sec)	4.611	0.152	0.258	0.258	0	0.207
elapsed time (sec)	86.082	82.286	82.586	82.593	82.284	82.4
average fetch time (msec)	15.643	16.576	15.929	16.083	16.36	16.349
average disk utilization	0.82	0.59	0.4	0.3	0.24	0.2
Batch size 80						
fetches	4698	5900	6307	6279	6140	5992
driver time (sec)	2.349	2.95	3.1535	3.1395	3.07	2.996
stall time (sec)	3.994	0.712	0.758	0.673	0.129	0.487
elapsed time (sec)	85.559	82.878	83.127	83.028	82.415	82.699
average fetch time (msec)	15.032	16.431	15.682	15.941	16.042	16.047
average disk utilization	0.83	0.58	0.4	0.3	0.24	0.19
Batch size 160						
fetches	4890	5940	6387	6270	6090	6176
driver time (sec)	2.445	2.97	3.1935	3.135	3.045	3.088
stall time (sec)	3.774	1.591	1.522	1.232	0.424	0.922
elapsed time (sec)	85.435	83.777	83.931	83.583	82.685	83.226
average fetch time (msec)	14.506	16.232	15.549	15.557	15.763	15.713
average disk utilization	0.83	0.58	0.39	0.29	0.23	0.19

Table A.37: Aggressive performance as a function of batch size on the postgres-select trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Batch size 4											
fetches	3085	3085	3085	3085	3085	3220	3633	3937	3902	3852	3731
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.61	1.8165	1.9685	1.951	1.926	1.8655
stall time (sec)	39.507	13.89	5.578	1.786	0.177	0.013	0.011	0.009	0.005	0.001	0
elapsed time (sec)	52.527	26.91	18.598	14.806	13.197	13.101	13.305	13.455	13.434	13.405	13.343
average fetch time (msec)	16.582	16.121	15.884	15.833	15.544	15.172	15.412	15.274	14.797	14.8	15.155
average disk utilization	0.97	0.92	0.88	0.82	0.73	0.62	0.6	0.56	0.43	0.35	0.26
Batch size 8											
fetches	3085	3085	3085	3085	3085	3317	3826	4068	3975	3854	3731
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.6585	1.913	2.034	1.9875	1.927	1.8655
stall time (sec)	37.572	13.285	4.968	1.198	0.005	0.001	0	0	0	0	0
elapsed time (sec)	50.592	26.305	17.988	14.218	13.025	13.137	13.391	13.512	13.465	13.405	13.343
average fetch time (msec)	15.972	15.679	15.45	15.156	15.113	15.036	15.221	14.961	14.501	14.653	14.895
average disk utilization	0.97	0.92	0.88	0.82	0.72	0.63	0.62	0.56	0.43	0.35	0.26
Batch size 16											
fetches	3085	3085	3085	3085	3286	3563	3989	4158	3975	3854	3731
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.643	1.7815	1.9945	2.079	1.9875	1.927	1.8655
stall time (sec)	35.953	12.022	4.149	0.844	0	0	0	0	0	0	0
elapsed time (sec)	48.973	25.042	17.169	13.864	13.121	13.259	13.472	13.557	13.465	13.405	13.343
average fetch time (msec)	15.468	15.047	14.865	14.55	13.923	14.605	14.883	14.47	14.155	14.44	14.675
average disk utilization	0.97	0.93	0.89	0.81	0.7	0.65	0.63	0.55	0.42	0.35	0.26
Batch size 40											
fetches	3085	3085	3085	3085	3498	3947	4207	4218	4059	3933	3741
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.749	1.9735	2.1035	2.109	2.0295	1.9665	1.8705
stall time (sec)	33.015	10.772	3.517	0.11	0.008	0.133	0.185	0.13	0	0.038	0
elapsed time (sec)	46.035	23.792	16.537	13.13	13.235	13.584	13.766	13.717	13.507	13.482	13.348
average fetch time (msec)	14.695	14.173	13.95	13.619	13.303	14.127	14.887	14.086	13.765	13.766	13.85
average disk utilization	0.98	0.92	0.87	0.8	0.7	0.68	0.65	0.54	0.41	0.33	0.24
Batch size 80											
fetches	3085	3085	3085	3222	3873	4118	4331	4157	4061	4100	3741
driver time (sec)	1.5425	1.5425	1.5425	1.611	1.9365	2.059	2.1655	2.0785	2.0305	2.05	1.8705
stall time (sec)	30.691	9.611	2.918	0.584	0.358	0.568	0.57	0.439	0	0.147	0
elapsed time (sec)	43.711	22.631	15.938	13.673	13.772	14.105	14.213	13.995	13.508	13.675	13.348
average fetch time (msec)	13.985	13.567	13.137	12.899	13.015	14.239	14.299	13.683	13.409	13.434	13.109
average disk utilization	0.99	0.92	0.85	0.76	0.73	0.69	0.62	0.51	0.4	0.34	0.23
Batch size 160											
fetches	3085	3085	3388	3762	4126	4355	4340	4237	3933	3820	3681
driver time (sec)	1.5425	1.5425	1.694	1.881	2.063	2.1775	2.17	2.1185	1.9665	1.91	1.8405
stall time (sec)	28.957	8.422	3.301	1.369	0.888	1.044	0.968	0.831	0.048	0.182	0
elapsed time (sec)	41.977	21.442	16.473	14.728	14.429	14.699	14.616	14.427	13.492	13.57	13.318
average fetch time (msec)	13.296	12.704	12.173	12.398	12.972	13.751	13.798	12.985	12.565	12.885	12.883
average disk utilization	0.98	0.91	0.83	0.79	0.74	0.68	0.59	0.48	0.37	0.3	0.22

Table A.38: Aggressive performance as a function of batch size on the xds trace.

Disks	1	2	3	4	5	6
Batch size 4						
fetches	5858	6939	6023	9564	8040	10098
driver time (sec)	2.929	3.4695	3.0115	4.782	4.02	5.049
stall time (sec)	32.888	0.162	1.06	0.02	0.018	0.015
elapsed time (sec)	65.896	33.71	34.15	34.881	34.117	35.143
average fetch time (msec)	10.977	7.635	14.623	9.75	15.515	10.778
average disk utilization	0.98	0.79	0.86	0.67	0.73	0.52
Batch size 8						
fetches	5862	7200	6202	9827	8068	10215
driver time (sec)	2.931	3.6	3.101	4.9135	4.034	5.1075
stall time (sec)	32.197	0.077	0.59	0.071	0.065	0.055
elapsed time (sec)	65.207	33.756	33.77	35.063	34.178	35.241
average fetch time (msec)	10.914	7.583	14.355	9.788	15.45	10.711
average disk utilization	0.98	0.81	0.88	0.69	0.73	0.52
Batch size 16						
fetches	5868	7522	6306	9831	8312	10124
driver time (sec)	2.934	3.761	3.153	4.9155	4.156	5.062
stall time (sec)	31.504	0.192	0.205	0.129	0.133	0.076
elapsed time (sec)	64.517	34.032	33.437	35.123	34.368	35.217
average fetch time (msec)	10.854	7.564	14.151	9.801	15.454	10.6
average disk utilization	0.99	0.84	0.89	0.69	0.75	0.51
Batch size 40						
fetches	5890	7778	6563	9929	8418	10353
driver time (sec)	2.945	3.889	3.2815	4.9645	4.209	5.1765
stall time (sec)	30.61	0.337	0.356	0.232	0.312	0.18
elapsed time (sec)	63.634	34.305	33.716	35.275	34.6	35.435
average fetch time (msec)	10.745	7.496	14.101	9.92	15.441	10.63
average disk utilization	0.99	0.85	0.91	0.7	0.75	0.52
Batch size 80						
fetches	5925	8126	6838	10150	8789	10461
driver time (sec)	2.9625	4.063	3.419	5.075	4.3945	5.2305
stall time (sec)	30.667	0.507	0.613	0.399	0.582	0.525
elapsed time (sec)	63.708	34.649	34.111	35.553	35.055	35.834
average fetch time (msec)	10.711	7.386	14.051	9.957	15.258	10.584
average disk utilization	1	0.87	0.94	0.71	0.77	0.51
Batch size 160						
fetches	6005	8198	7519	10300	8601	10488
driver time (sec)	3.0025	4.099	3.7595	5.15	4.3005	5.244
stall time (sec)	31.271	0.951	1.998	1.331	1.608	0.978
elapsed time (sec)	64.352	35.129	35.836	35.56	35.987	36.301
average fetch time (msec)	10.707	7.447	13.692	9.715	15.213	10.363
average disk utilization	1	0.87	0.96	0.68	0.73	0.5

A.6 Performance data: varying *reverse aggressive*'s parameters

This section contains the performance data for *reverse aggressive* with varying batch sizes and fetch time estimates. For brevity, only the elapsed times are shown.

Table A.39: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the dinero trace.

Disks	1	2	3	4	5	6
Fetch time 4						
Batch size 4	105.932	106.013	106.992	107.961	107.959	107.957
Batch size 8	105.946	106.013	106.991	107.953	107.949	107.945
Batch size 16	105.976	106.093	106.988	107.944	107.944	107.944
Batch size 40	106.03	106.348	107.021	107.944	107.944	107.944
Batch size 80	106.283	106.798	107.468	107.944	107.944	107.947
Batch size 160	106.825	107.719	108.033	108.044	107.944	107.954
Fetch time 8						
Batch size 4	105.931	105.949	105.979	106.013	106.502	106.99
Batch size 8	105.944	105.981	106.023	106.119	106.502	106.99
Batch size 16	105.972	106.094	106.13	106.335	106.512	107.004
Batch size 40	106.01	106.332	106.476	106.891	107.282	107.542
Batch size 80	106.22	106.716	107.104	107.805	107.944	107.944
Batch size 160	106.708	107.425	108.11	108.148	107.944	107.954
Fetch time 16						
Batch size 4	105.929	105.946	105.978	105.986	106.047	:06.06
Batch size 8	105.945	105.977	106.028	106.09	106.146	106.191
Batch size 16	105.976	106.093	106.156	106.233	106.344	106.459
Batch size 40	105.975	106.32	106.478	106.743	107.241	107.499
Batch size 80	106.181	106.716	107.227	107.685	107.939	107.944
Batch size 160	106.684	107.425	108.149	108.059	107.944	108.014
Fetch time 32						
Batch size 4	105.927	105.945	105.978	105.981	106.047	106.069
Batch size 8	105.942	105.977	106.064	106.091	106.134	106.163
Batch size 16	105.974	106.093	106.161	106.253	106.329	106.402
Batch size 40	105.982	106.288	106.508	106.783	107.107	107.454
Batch size 80	106.15	106.716	107.371	107.659	107.935	107.948
Batch size 160	106.612	107.398	108.159	108.074	107.944	108.014
Fetch time 64						
Batch size 4	105.927	105.941	105.972	105.978	106.047	106.063
Batch size 8	105.941	105.977	106.025	106.106	106.139	106.171
Batch size 16	105.969	106.089	106.17	106.203	106.302	106.369
Batch size 40	105.987	106.304	106.464	106.749	107.011	107.513
Batch size 80	106.15	106.716	107.268	107.594	107.907	107.944
Batch size 160	106.628	107.407	108.153	108.163	107.944	107.944
Fetch time 128						
Batch size 4	105.927	105.941	105.972	105.97	106.01	106.063
Batch size 8	105.941	105.969	106.017	106.09	106.135	106.171
Batch size 16	105.969	106.089	106.17	106.336	106.314	106.419
Batch size 40	105.987	106.312	106.539	106.689	106.994	107.484
Batch size 80	106.15	106.716	107.27	107.603	107.932	107.952
Batch size 160	106.66	107.425	108.154	108.164	107.944	107.984

Table A.40: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the cscope1 trace.

Disks	1	2	3	4	5	6
Fetch time 4						
Batch size 4	29.884	27.485	28.328	29.219	29.217	29.215
Batch size 8	29.509	27.529	28.328	29.211	29.207	29.203
Batch size 16	29.42	27.589	28.327	29.202	29.202	29.202
Batch size 40	29.339	27.807	28.351	29.202	29.202	29.202
Batch size 80	29.098	28.238	28.778	29.202	29.202	29.202
Batch size 160	29.894	29.255	29.404	29.348	29.202	29.202
Fetch time 8						
Batch size 4	30.199	27.47	27.489	27.526	27.854	28.326
Batch size 8	30.072	27.486	27.532	27.593	27.851	28.326
Batch size 16	29.987	27.536	27.619	27.757	27.872	28.349
Batch size 40	29.479	27.678	27.927	28.231	28.528	28.937
Batch size 80	28.921	28.015	28.522	29.065	29.195	29.202
Batch size 160	29.792	29.038	29.351	29.438	29.202	29.202
Fetch time 16						
Batch size 4	30.379	27.461	27.477	27.498	27.515	27.541
Batch size 8	30.34	27.51	27.517	27.556	27.603	27.661
Batch size 16	30.177	27.525	27.639	27.682	27.78	27.894
Batch size 40	29.683	27.756	27.869	28.066	28.362	28.664
Batch size 80	29.105	28.104	28.463	28.906	29.181	29.202
Batch size 160	30.051	29.045	29.377	29.318	29.202	29.202
Fetch time 32						
Batch size 4	30.499	27.457	27.471	27.513	27.528	27.527
Batch size 8	30.423	27.507	27.513	27.544	27.592	27.673
Batch size 16	30.351	27.51	27.605	27.673	27.767	27.888
Batch size 40	30.048	27.725	27.954	28.159	28.374	28.675
Batch size 80	29.672	28.072	28.618	28.907	29.188	29.202
Batch size 160	30.773	29.036	29.393	29.455	29.202	29.202
Fetch time 64						
Batch size 4	30.544	27.453	27.471	27.506	27.518	27.515
Batch size 8	30.465	27.48	27.513	27.563	27.598	27.662
Batch size 16	30.319	27.513	27.61	27.654	27.743	27.855
Batch size 40	30.171	27.74	27.91	28.054	28.459	28.722
Batch size 80	30.055	28.087	28.521	28.904	29.185	29.202
Batch size 160	30.259	29.037	29.295	29.273	29.202	29.202
Fetch time 128						
Batch size 4	30.559	27.453	27.465	27.505	27.517	27.515
Batch size 8	30.521	27.48	27.506	27.548	27.598	27.657
Batch size 16	30.433	27.513	27.608	27.745	27.753	27.874
Batch size 40	30.224	27.752	27.998	28.075	28.4	28.759
Batch size 80	30.088	28.111	28.529	28.898	29.189	29.202
Batch size 160	30.169	29.053	29.324	29.236	29.202	29.202

Table A.41: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the cscope2 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 4											
Batch size 4	73.646	76.154	58.632	46.844	42.027	42.025	42.023	42.021	42.017	42.028	42.02
Batch size 8	68.713	73.32	56.656	45.18	42.017	42.013	42.012	42.02	42.012	42.019	42.018
Batch size 16	64.597	69.609	54.03	42.527	42.012	42.012	42.012	42.012	42.012	42.012	42.014
Batch size 40	60.204	65.388	50.161	42.228	42.056	42.167	42.132	42.118	42.012	42.167	42.063
Batch size 80	58.676	61.528	46.006	42.612	42.377	42.467	42.295	42.303	42.242	42.397	42.414
Batch size 160	58.824	58.068	43.961	43.221	43.178	43.163	43.107	43.137	43.002	42.705	42.431
Fetch time 8											
Batch size 4	66.301	53.815	49.939	45.845	42.027	42.025	42.023	42.021	42.017	42.028	42.02
Batch size 8	65.976	52.196	47.404	44.208	42.017	42.013	42.012	42.02	42.012	42.019	42.018
Batch size 16	63.699	50.01	44.751	42.175	42.012	42.012	42.012	42.012	42.012	42.012	42.014
Batch size 40	61.436	49.007	43.106	42.227	42.056	42.167	42.132	42.118	42.012	42.167	42.063
Batch size 80	59.443	48.204	42.297	42.612	42.377	42.467	42.295	42.303	42.242	42.397	42.414
Batch size 160	59.338	49.797	42.507	43.221	43.178	43.163	43.108	43.137	43.002	42.705	42.431
Fetch time 16											
Batch size 4	65.726	55.805	46.45	41.044	40.724	41.196	41.659	41.944	42.014	42.028	42.02
Batch size 8	66.026	53.985	45.297	40.555	40.723	41.195	41.666	41.983	42.012	42.019	42.018
Batch size 16	64.078	52.309	44.325	40.254	40.734	41.215	41.694	42.005	42.012	42.012	42.014
Batch size 40	61.326	51.027	43.843	40.661	40.83	41.439	41.892	42.118	42.012	42.167	42.063
Batch size 80	59.733	48.032	42.27	41.451	41.396	41.843	42.191	42.303	42.242	42.397	42.413
Batch size 160	58.255	48.498	42.151	42.79	43.061	43.116	43.106	43.123	43.002	42.705	42.431
Fetch time 32											
Batch size 4	66.369	55.811	47.22	42.32	40.412	40.16	40.163	40.251	40.722	41.209	41.918
Batch size 8	66.538	54.272	46.13	41.626	40.176	40.197	40.218	40.265	40.734	41.218	41.953
Batch size 16	64.857	52.296	44.816	40.717	40.258	40.309	40.344	40.375	40.77	41.255	41.994
Batch size 40	61.57	50.761	43.215	40.604	40.542	40.8	40.815	40.925	41.024	41.565	42.063
Batch size 80	60.573	48.144	41.63	41.277	41.251	41.61	41.587	41.846	42.048	42.355	42.415
Batch size 160	59.52	47.419	42.172	42.502	42.862	43.117	42.985	42.976	43.202	43.14	42.431
Fetch time 64											
Batch size 4	66.686	55.88	47.224	42.28	40.322	40.162	40.164	40.176	40.18	40.214	40.265
Batch size 8	66.943	54.08	46.155	41.415	40.185	40.191	40.201	40.251	40.263	40.312	40.39
Batch size 16	65.161	52.358	44.88	40.69	40.255	40.279	40.321	40.363	40.436	40.503	40.655
Batch size 40	62.331	50.149	43.592	40.636	40.549	40.706	40.763	40.865	40.942	41.286	41.539
Batch size 80	60.58	47.05	41.959	41.281	41.211	41.485	41.504	41.703	41.978	42.323	42.296
Batch size 160	59.045	47.689	42.042	42.517	42.849	43.065	43.107	43.159	43.189	43.294	42.313
Fetch time 128											
Batch size 4	66.981	56.125	47.224	42.265	40.322	40.158	40.164	40.177	40.19	40.229	40.236
Batch size 8	67.219	54.329	46.125	41.412	40.187	40.196	40.218	40.237	40.255	40.298	40.367
Batch size 16	65.627	52.364	44.691	40.759	40.244	40.267	40.361	40.346	40.394	40.479	40.617
Batch size 40	61.941	50.631	43.293	40.592	40.501	40.68	40.833	40.915	40.876	41.199	41.475
Batch size 80	60.618	48.25	41.506	41.355	41.268	41.476	41.473	41.665	41.959	42.289	42.339
Batch size 160	59.294	46.826	41.969	42.696	42.856	43.096	43.044	42.959	43.168	43.314	42.311

Table A.42: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the cscope3 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 4											
Batch size 4	120.374	106.657	86.841	82.28	81.905	81.903	81.901	81.899	81.895	81.906	81.898
Batch size 8	116.187	102.762	83.583	82.011	81.895	81.891	81.89	81.898	81.89	81.897	81.896
Batch size 16	112.783	96.87	82.994	81.957	81.89	81.926	81.89	81.89	81.89	81.968	81.892
Batch size 40	107.785	91.908	82.549	82.224	82.184	82.154	81.964	81.993	82.133	82.226	81.999
Batch size 80	105.834	87.809	82.778	82.593	82.617	82.408	82.259	82.375	82.433	82.482	82.347
Batch size 160	104.065	85.424	83.448	83.379	83.458	83.014	83.052	83.083	82.751	82.653	82.354
Fetch time 8											
Batch size 4	118.714	99.498	82.058	82.09	81.905	81.903	81.901	81.899	81.895	81.906	81.898
Batch size 8	117.231	96.441	81.642	81.832	81.895	81.891	81.89	81.898	81.89	81.897	81.896
Batch size 16	114.398	93.24	81.419	81.797	81.89	81.926	81.89	81.89	81.89	81.968	81.892
Batch size 40	109.654	90.407	81.011	82.111	82.184	82.154	81.964	81.993	82.133	82.226	81.999
Batch size 80	105.912	87.902	81.411	82.556	82.617	82.408	82.259	82.375	82.433	82.482	82.347
Batch size 160	105.316	85.933	82.611	83.379	83.457	83.014	83.052	83.083	82.751	82.653	82.354
Fetch time 16											
Batch size 4	120.959	99.258	86.509	82.047	80.095	80.317	81.024	81.7	81.895	81.906	81.898
Batch size 8	119.269	96.896	85.597	81.219	80.072	80.316	81.025	81.72	81.89	81.897	81.896
Batch size 16	116.419	92.958	84.417	80.524	80.17	80.371	81.047	81.738	81.89	81.968	81.892
Batch size 40	111.804	89.826	82.373	80.647	80.737	80.806	81.203	81.91	82.133	82.226	81.999
Batch size 80	107.929	86.972	81.381	81.365	81.597	81.633	81.846	82.362	82.431	82.482	82.347
Batch size 160	106.978	84.24	82.473	82.839	83.309	82.926	83.044	83.094	82.751	82.653	82.354
Fetch time 32											
Batch size 4	121.953	100.686	86.669	82.028	80.113	80.043	80.05	80.058	80.082	80.33	81.644
Batch size 8	121.183	98.076	85.666	81.279	80.056	80.08	80.091	80.127	80.168	80.341	81.668
Batch size 16	118.298	94.376	84.202	80.741	80.13	80.202	80.214	80.259	80.357	80.545	81.709
Batch size 40	112.982	90.68	82.261	80.612	80.66	80.742	80.645	80.757	81.137	81.409	81.92
Batch size 80	109.193	86.671	81.336	81.293	81.505	81.491	81.592	81.856	82.252	82.319	82.295
Batch size 160	108.381	84.039	82.41	82.678	83.19	82.925	83.065	83.095	82.857	82.675	82.324
Fetch time 64											
Batch size 4	122.666	101.556	87.388	81.982	80.065	80.035	80.038	80.051	80.065	80.094	80.131
Batch size 8	121.796	98.68	86.101	81.224	80.047	80.07	80.078	80.117	80.145	80.185	80.266
Batch size 16	118.707	94.765	84.931	80.619	80.111	80.186	80.195	80.248	80.319	80.452	80.535
Batch size 40	113.545	91.279	82.495	80.602	80.615	80.661	80.584	80.723	81.03	81.328	81.463
Batch size 80	108.795	86.447	81.422	81.186	81.401	81.361	81.526	81.765	82.225	82.32	82.2
Batch size 160	106.666	84.729	82.374	82.613	83.161	82.904	82.979	83.022	82.669	82.372	82.294
Fetch time 128											
Batch size 4	123.14	102.073	87.561	81.879	80.069	80.032	80.054	80.051	80.065	80.109	80.111
Batch size 8	122.151	98.879	86	81.045	80.049	80.07	80.084	80.111	80.12	80.174	80.244
Batch size 16	118.865	94.58	84.897	80.559	80.111	80.176	80.177	80.207	80.274	80.445	80.495
Batch size 40	113.657	91.243	82.453	80.625	80.587	80.641	80.562	80.67	80.96	81.225	81.426
Batch size 80	109.424	87.42	81.338	81.213	81.425	81.377	81.401	81.753	82.214	82.361	82.153
Batch size 160	107.025	84.923	82.402	82.926	83.243	82.917	83.057	83.083	82.857	82.548	82.339

Table A.43: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the glimpse trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 4											
Batch size 4	120.334	85.098	65.108	53.355	47.429	44.047	43.626	43.624	43.62	43.62	43.615
Batch size 8	117.591	83.434	63.074	51.409	46.198	43.674	43.615	43.623	43.615	43.617	43.615
Batch size 16	114.802	81.065	60.735	49.93	44.488	43.615	43.615	43.615	43.615	43.615	43.615
Batch size 40	111.624	76.967	57.331	46.21	44.075	43.615	43.615	43.615	43.615	43.615	43.615
Batch size 80	109.285	71.402	53.632	44.116	43.615	43.665	43.649	43.735	43.615	43.794	43.665
Batch size 160	104.802	67.211	51.041	44.23	44.131	44.265	44.2	44.229	43.615	43.979	43.655
Fetch time 8											
Batch size 4	107.913	73.03	64.973	53.265	47.429	44.047	43.626	43.624	43.62	43.62	43.615
Batch size 8	105.957	71.325	62.994	51.409	46.198	43.674	43.615	43.623	43.615	43.617	43.615
Batch size 16	103.948	68.957	60.837	49.93	44.488	43.615	43.615	43.615	43.615	43.615	43.615
Batch size 40	101.399	65.479	57.211	46.21	44.075	43.615	43.615	43.615	43.615	43.615	43.615
Batch size 80	99.3	60.248	53.496	44.116	43.615	43.665	43.649	43.735	43.615	43.794	43.665
Batch size 160	97.201	60.57	51.011	44.23	44.131	44.265	44.2	44.229	43.615	43.979	43.655
Fetch time 16											
Batch size 4	105.481	66.119	53.034	46.815	45.171	44.013	43.623	43.623	43.62	43.62	43.615
Batch size 8	103.729	65.143	51.537	45.8	44.713	43.652	43.613	43.623	43.615	43.617	43.615
Batch size 16	101.83	63.657	50.158	45.06	44.049	43.609	43.614	43.615	43.615	43.615	43.615
Batch size 40	99.838	60.979	48.998	43.969	43.77	43.614	43.615	43.615	43.615	43.615	43.615
Batch size 80	96.914	58.512	47.502	43.282	43.394	43.665	43.649	43.735	43.615	43.794	43.665
Batch size 160	94.952	58.988	48.418	43.967	44.053	44.264	44.2	44.229	43.615	43.979	43.655
Fetch time 32											
Batch size 4	106.155	66.602	53.799	47.221	44.527	42.727	42.336	42.609	43.235	43.593	43.614
Batch size 8	104.101	65.488	52.235	46.245	44.08	42.394	42.358	42.623	43.252	43.605	43.614
Batch size 16	102.238	63.442	50.491	45.059	43.337	42.334	42.416	42.639	43.29	43.612	43.614
Batch size 40	98.747	61.039	48.803	44.041	42.977	42.54	42.66	42.794	43.391	43.615	43.614
Batch size 80	96.797	58.354	47.504	43.707	42.701	42.907	43.076	43.344	43.565	43.794	43.651
Batch size 160	95.56	58.234	48.023	43.924	43.899	44.209	44.195	44.209	43.615	44.024	43.655
Fetch time 64											
Batch size 4	106.528	67.17	53.854	47.269	44.426	42.57	42.142	42.17	42.224	42.289	42.61
Batch size 8	104.416	66.029	52.415	46.318	43.831	42.257	42.159	42.198	42.262	42.339	42.637
Batch size 16	102.302	63.946	50.393	45.07	43.04	42.158	42.222	42.272	42.36	42.479	42.708
Batch size 40	99.538	60.855	48.945	43.91	42.873	42.303	42.509	42.542	42.717	42.911	43.22
Batch size 80	96.534	58.395	47.984	43.436	42.532	42.802	43.041	43.248	43.453	43.732	43.611
Batch size 160	94.083	58.341	48.649	44.205	43.903	44.172	44.197	44.102	43.614	44.067	43.614
Fetch time 128											
Batch size 4	106.633	67.215	54.05	47.374	44.304	42.508	42.055	42.08	42.096	42.133	42.205
Batch size 8	104.446	66.059	52.287	46.112	43.782	42.211	42.078	42.114	42.136	42.186	42.28
Batch size 16	102.287	63.773	50.739	45.159	43.046	42.118	42.134	42.181	42.246	42.319	42.452
Batch size 40	99.762	60.971	48.726	43.902	42.849	42.377	42.315	42.416	42.577	42.759	43.17
Batch size 80	96.544	58.679	47.561	43.747	42.526	42.81	42.818	43.123	43.407	43.734	43.612
Batch size 160	94.895	60.33	48.078	44.09	44.062	44.092	44.094	44.014	43.611	44.067	43.612

Table A.44: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the ld trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 4											
Batch size 4	26.128	18.833	17.411	15.52	13.862	12.667	11.554	10.437	10.249	10.26	10.252
Batch size 8	26.651	18.755	16.815	15.207	13.373	11.861	10.74	10.252	10.244	10.251	10.25
Batch size 16	26.177	17.831	16.035	14.473	12.608	11.452	10.244	10.244	10.244	10.244	10.246
Batch size 40	25.345	16.858	15.096	13.367	11.83	10.809	10.353	10.318	10.244	10.338	10.278
Batch size 80	24.775	16.712	14.936	13.224	11.23	10.582	10.591	10.577	10.448	10.597	10.569
Batch size 160	24.347	16.47	14.807	12.826	11.44	11.23	11.393	11.411	11.29	11.006	10.571
Fetch time 8											
Batch size 4	26.157	17.987	14.787	13.436	12.555	12.183	11.417	10.437	10.249	10.26	10.252
Batch size 8	26.642	17.867	14.172	12.895	12.013	11.394	10.74	10.252	10.244	10.251	10.25
Batch size 16	26.027	16.935	13.557	12.131	11.382	10.928	10.244	10.244	10.244	10.244	10.246
Batch size 40	25.348	16.289	13.392	11.525	10.767	10.666	10.353	10.318	10.244	10.338	10.278
Batch size 80	24.95	16.012	13.046	11.622	10.91	10.654	10.646	10.577	10.448	10.597	10.569
Batch size 160	24.377	16.05	13.599	12.282	11.615	11.358	11.393	11.411	11.29	11.006	10.571
Fetch time 16											
Batch size 4	26.082	18.02	14.686	12.761	11.182	10.599	10.147	9.886	9.959	10.138	10.252
Batch size 8	26.591	17.75	14.203	12.176	11.07	10.516	10.017	9.868	9.971	10.151	10.25
Batch size 16	25.982	17.166	13.636	11.806	10.683	10.341	9.927	9.854	10.01	10.18	10.246
Batch size 40	25.223	16.171	13.331	11.671	10.624	10.42	10.15	9.978	10.124	10.335	10.278
Batch size 80	24.825	15.921	13.057	11.805	11	10.664	10.587	10.61	10.46	10.597	10.569
Batch size 160	24.462	15.972	13.897	12.294	12.024	11.948	11.858	11.301	11.59	10.991	10.586
Fetch time 32											
Batch size 4	26.029	17.934	14.696	12.701	11.163	10.571	10.128	9.861	9.676	9.683	9.793
Batch size 8	26.351	17.837	14.292	12.17	10.954	10.434	10.118	9.816	9.678	9.699	9.817
Batch size 16	25.962	16.986	13.582	11.772	10.711	10.344	10.052	9.835	9.768	9.754	9.879
Batch size 40	25.27	16.282	13.347	11.847	10.747	10.447	10.341	10.008	9.995	10.052	10.129
Batch size 80	24.9	16.073	13.288	12.003	11.215	10.837	10.781	10.647	10.382	10.572	10.577
Batch size 160	24.435	16.01	13.66	12.866	12.324	12.356	12.229	11.772	11.535	11.097	10.826
Fetch time 64											
Batch size 4	26.029	17.934	14.68	12.656	11.224	10.504	10.166	9.834	9.692	9.767	9.677
Batch size 8	26.351	17.762	14.23	12.215	11.08	10.474	10.098	9.896	9.709	9.737	9.713
Batch size 16	26.244	17.018	13.503	11.91	10.825	10.375	10.111	10.009	9.768	9.746	9.794
Batch size 40	25.167	16.238	13.364	11.949	10.682	10.548	10.358	10.204	9.931	10.034	10.106
Batch size 80	24.888	16.188	13.216	12.046	11.189	10.97	10.776	10.645	10.605	10.708	10.67
Batch size 160	24.392	15.953	13.671	12.775	12.323	12.384	12.119	12.194	11.985	11.542	11.076
Fetch time 128											
Batch size 4	26.029	17.934	14.68	12.656	11.285	10.481	10.165	9.903	9.677	9.767	9.678
Batch size 8	26.351	17.762	14.278	12.19	11.05	10.442	10.129	9.862	9.701	9.735	9.714
Batch size 16	26.244	16.973	13.547	11.743	10.779	10.301	10.188	9.871	9.774	9.759	9.796
Batch size 40	25.219	16.114	13.346	11.759	10.749	10.471	10.515	10.203	10.178	10.06	10.088
Batch size 80	24.771	16.056	12.999	11.9	11.462	11.147	11.111	11.12	10.722	10.721	10.447
Batch size 160	24.377	16.148	13.651	12.722	12.605	12.135	12.557	12.579	12.059	11.549	10.615

Table A.45: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the postgres-join trace.

Disks	1	2	3	4	5	6
Fetch time 4						
Batch size 4	92.487	81.692	81.991	81.993	81.992	81.99
Batch size 8	90.455	81.691	81.988	81.986	81.982	81.978
Batch size 16	88.578	81.69	81.976	81.977	81.977	81.977
Batch size 40	85.763	81.871	82.253	82.25	81.977	82.234
Batch size 80	85.166	82.485	82.732	82.68	82.106	82.544
Batch size 160	85.002	83.489	83.497	83.284	82.401	83.086
Fetch time 8						
Batch size 4	92.487	81.163	81.359	81.694	81.965	81.986
Batch size 8	90.455	81.165	81.358	81.693	81.974	81.976
Batch size 16	88.578	81.169	81.357	81.7	81.975	81.975
Batch size 40	85.763	81.364	81.665	82.02	81.976	82.233
Batch size 80	85.166	82.024	82.204	82.525	82.105	82.543
Batch size 160	84.986	83.106	83.232	83.242	82.4	83.085
Fetch time 16						
Batch size 4	92.487	81.164	81.166	81.173	81.224	81.365
Batch size 8	90.455	81.166	81.17	81.177	81.224	81.362
Batch size 16	88.578	81.168	81.177	81.192	81.238	81.354
Batch size 40	85.763	81.363	81.514	81.568	81.352	81.71
Batch size 80	85.165	81.998	82.126	82.186	81.68	82.257
Batch size 160	84.99	83.067	83.165	83.141	82.342	83.04
Fetch time 32						
Batch size 4	92.487	81.164	81.165	81.17	81.173	81.177
Batch size 8	90.455	81.164	81.167	81.176	81.18	81.187
Batch size 16	88.578	81.164	81.171	81.193	81.208	81.226
Batch size 40	85.765	81.36	81.497	81.565	81.312	81.633
Batch size 80	85.165	81.984	82.073	82.116	81.689	82.219
Batch size 160	84.984	83.02	83.063	83.104	82.347	82.961
Fetch time 64						
Batch size 4	92.487	81.164	81.164	81.169	81.17	81.176
Batch size 8	90.455	81.167	81.166	81.172	81.175	81.18
Batch size 16	88.578	81.169	81.168	81.184	81.199	81.213
Batch size 40	85.765	81.365	81.494	81.523	81.293	81.604
Batch size 80	85.165	81.976	82.051	82.086	81.627	82.155
Batch size 160	84.985	83.015	83.014	83.091	82.346	83.054
Fetch time 128						
Batch size 4	92.487	81.165	81.166	81.172	81.171	81.175
Batch size 8	90.455	81.165	81.17	81.177	81.177	81.178
Batch size 16	88.578	81.169	81.172	81.18	81.195	81.213
Batch size 40	85.765	81.352	81.498	81.518	81.289	81.577
Batch size 80	85.168	81.982	82.059	82.082	81.617	82.155
Batch size 160	84.988	83.026	83.075	83.043	82.347	83.032

Table A.46: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the postgres-select trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 4											
Batch size 4	52.54	26.927	20.823	16.389	13.787	13.277	13.275	13.273	13.269	13.28	13.272
Batch size 8	50.627	26.314	20.25	15.712	13.294	13.265	13.264	13.272	13.264	13.271	13.27
Batch size 16	49.017	25.066	19.493	14.959	13.264	13.264	13.264	13.264	13.264	13.264	13.266
Batch size 40	46.106	23.816	18.577	13.859	13.272	13.412	13.508	13.476	13.264	13.341	13.287
Batch size 80	43.782	22.778	17.716	13.878	13.622	13.906	13.88	13.821	13.313	13.618	13.641
Batch size 160	41.995	22.471	16.889	14.707	14.281	14.368	14.496	14.338	13.832	13.806	13.656
Fetch time 8											
Batch size 4	52.54	26.925	18.686	14.807	13.157	13.248	13.27	13.271	13.268	13.28	13.271
Batch size 8	50.606	26.313	18.011	14.236	13.131	13.246	13.263	13.27	13.263	13.271	13.269
Batch size 16	48.987	25.066	17.278	13.882	13.146	13.259	13.263	13.262	13.263	13.264	13.265
Batch size 40	46.075	23.816	16.586	13.181	13.209	13.412	13.507	13.474	13.263	13.341	13.286
Batch size 80	43.782	22.724	16.002	13.749	13.611	13.907	13.879	13.819	13.312	13.618	13.641
Batch size 160	41.995	21.496	15.797	14.672	14.253	14.368	14.494	14.336	13.831	13.806	13.655
Fetch time 16											
Batch size 4	52.557	26.925	18.687	14.807	13.078	13.052	13.05	13.048	13.134	13.255	13.272
Batch size 8	50.627	26.314	18.011	14.236	13.044	13.04	13.039	13.047	13.147	13.26	13.27
Batch size 16	49.017	25.066	17.278	13.882	13.039	13.039	13.039	13.051	13.185	13.263	13.266
Batch size 40	46.106	23.816	16.586	13.164	13.047	13.191	13.326	13.343	13.254	13.34	13.287
Batch size 80	43.782	22.724	16.001	13.654	13.432	13.779	13.814	13.814	13.314	13.617	13.641
Batch size 160	41.995	21.496	16.398	14.681	14.293	14.345	14.494	14.301	13.827	13.805	13.641
Fetch time 32											
Batch size 4	52.535	26.921	18.681	14.801	13.074	13.05	13.049	13.048	13.044	13.055	13.054
Batch size 8	50.599	26.311	18.007	14.236	13.044	13.04	13.039	13.047	13.039	13.046	13.068
Batch size 16	48.98	25.066	17.278	13.882	13.039	13.039	13.039	13.039	13.039	13.039	13.118
Batch size 40	46.039	23.816	16.586	13.164	13.047	13.187	13.283	13.255	13.07	13.198	13.27
Batch size 80	43.711	22.724	16.001	13.654	13.406	13.746	13.791	13.754	13.309	13.615	13.617
Batch size 160	41.987	21.496	16.333	14.638	14.208	14.37	14.504	14.184	13.746	13.603	13.638
Fetch time 64											
Batch size 4	52.531	26.918	18.682	14.802	13.073	13.047	13.045	13.044	13.041	13.053	13.047
Batch size 8	50.595	26.309	18.008	14.232	13.038	13.035	13.038	13.047	13.039	13.046	13.045
Batch size 16	48.98	25.062	17.275	13.882	13.039	13.039	13.039	13.039	13.039	13.039	13.042
Batch size 40	46.039	23.812	16.586	13.164	13.047	13.187	13.284	13.251	13.064	13.199	13.275
Batch size 80	43.711	22.716	16.001	13.653	13.424	13.779	13.777	13.748	13.317	13.6	13.588
Batch size 160	41.987	21.497	15.924	14.636	14.295	14.343	14.504	14.22	13.713	13.873	13.621
Fetch time 128											
Batch size 4	52.529	26.914	18.676	14.798	13.07	13.045	13.045	13.043	13.04	13.053	13.047
Batch size 8	50.595	26.309	18.003	14.228	13.036	13.033	13.034	13.042	13.036	13.044	13.045
Batch size 16	48.98	25.062	17.27	13.874	13.032	13.037	13.035	13.039	13.039	13.039	13.042
Batch size 40	46.039	23.812	16.578	13.158	13.044	13.187	13.283	13.251	13.064	13.22	13.243
Batch size 80	43.711	22.716	15.993	13.649	13.397	13.733	13.801	13.774	13.266	13.609	13.622
Batch size 160	41.987	21.492	16.415	14.582	14.199	14.736	14.555	14.492	13.752	13.975	13.633

Table A.47: Reverse aggressive elapsed time as a function of fetch time estimate and batch size on the xds trace.

Disks	1	2	3	4	5	6
Fetch time 4						
Batch size 4	66.434	33.368	39.661	34.022	34.103	34.097
Batch size 8	65.548	33.295	38.575	34.089	34.162	34.155
Batch size 16	65.02	33.423	37.646	34.181	34.242	34.181
Batch size 40	64.155	33.608	37.314	34.331	34.419	34.302
Batch size 80	64.275	33.807	37.418	34.563	34.701	34.741
Batch size 160	65.044	34.446	37.36	35.585	35.9	35.358
Fetch time 8						
Batch size 4	66.426	33.35	34.334	33.23	33.538	33.767
Batch size 8	65.723	33.262	33.891	33.29	33.606	33.833
Batch size 16	64.971	33.302	33.285	33.38	33.704	33.876
Batch size 40	64.184	33.493	33.643	33.563	33.938	34.04
Batch size 80	64.185	33.728	33.934	33.934	34.315	34.533
Batch size 160	64.759	34.379	34.967	35.23	35.802	35.293
Fetch time 16						
Batch size 4	66.423	33.348	34.616	33.138	33.048	33.127
Batch size 8	65.689	33.2	34.235	33.175	33.119	33.202
Batch size 16	64.974	33.3	33.477	33.269	33.235	33.276
Batch size 40	64.171	33.495	33.473	33.479	33.528	33.546
Batch size 80	64.336	33.731	33.866	33.834	34.058	34.226
Batch size 160	64.519	34.361	34.822	35.164	35.741	35.163
Fetch time 32						
Batch size 4	66.423	33.348	34.534	33.125	33.044	33.105
Batch size 8	65.658	33.215	34.142	33.171	33.116	33.177
Batch size 16	64.967	33.293	33.57	33.257	33.227	33.251
Batch size 40	64.109	33.485	33.488	33.457	33.514	33.485
Batch size 80	64.092	33.792	33.859	33.844	33.966	34.173
Batch size 160	64.713	34.339	34.797	35.126	35.731	35.034
Fetch time 64						
Batch size 4	66.423	33.348	34.534	33.125	33.042	33.105
Batch size 8	65.658	33.215	34.151	33.171	33.115	33.17
Batch size 16	64.967	33.293	33.433	33.252	33.223	33.229
Batch size 40	64.18	33.48	33.468	33.431	33.471	33.478
Batch size 80	64.199	33.688	33.813	33.796	33.951	34.162
Batch size 160	64.789	34.278	34.853	35.168	35.739	35.193
Fetch time 128						
Batch size 4	66.423	33.348	34.534	33.125	33.042	33.105
Batch size 8	65.658	33.215	34.151	33.165	33.115	33.179
Batch size 16	64.967	33.293	33.433	33.259	33.225	33.239
Batch size 40	64.18	33.486	33.48	33.44	33.474	33.463
Batch size 80	64.26	33.692	33.826	33.778	33.915	34.159
Batch size 160	64.924	34.299	34.813	35.175	35.662	35.198

A.7 Performance data: varying *fixed horizon*'s horizon

This section contains the performance data for *fixed horizon* with varying values of the horizon.

Table A.48: Fixed horizon performance as a function of horizon on the dinero trace.

Disks	1	2	3	4	5	6
Horizon 16						
fetches	4716	4716	4716	4716	4716	4716
driver time (sec)	2.358	2.358	2.358	2.358	2.358	2.358
stall time (sec)	0.023	0.023	0.023	0.023	0.023	0.023
elapsed time (sec)	105.919	105.919	105.919	105.919	105.919	105.919
average fetch time (msec)	3.153	3.171	3.196	3.234	3.245	3.293
average disk utilization	0.14	0.071	0.047	0.036	0.029	0.024
Horizon 32						
fetches	4716	4716	4716	4716	4716	4716
driver time (sec)	2.358	2.358	2.358	2.358	2.358	2.358
stall time (sec)	0.022	0.022	0.022	0.022	0.022	0.022
elapsed time (sec)	105.918	105.918	105.918	105.918	105.918	105.918
average fetch time (msec)	3.145	3.182	3.201	3.241	3.259	3.294
average disk utilization	0.14	0.071	0.048	0.036	0.029	0.024
Horizon 64						
fetches	4789	4789	4789	4789	4789	4789
driver time (sec)	2.3945	2.3945	2.3945	2.3945	2.3945	2.3945
stall time (sec)	0.026	0.008	0.008	0.008	0.008	0.008
elapsed time (sec)	105.959	105.941	105.941	105.941	105.941	105.941
average fetch time (msec)	3.155	3.19	3.232	3.269	3.29	3.328
average disk utilization	0.14	0.072	0.049	0.037	0.03	0.025
Horizon 128						
fetches	5182	5182	5182	5182	5182	5182
driver time (sec)	2.591	2.591	2.591	2.591	2.591	2.591
stall time (sec)	0.249	0	0	0	0	0
elapsed time (sec)	106.378	106.129	106.129	106.129	106.129	106.129
average fetch time (msec)	3.171	3.208	3.256	3.286	3.32	3.37
average disk utilization	0.15	0.078	0.053	0.04	0.032	0.027
Horizon 256						
fetches	6005	6005	6005	6005	6005	6005
driver time (sec)	3.0025	3.0025	3.0025	3.0025	3.0025	3.0025
stall time (sec)	0.664	0	0.025	0	0	0
elapsed time (sec)	107.205	106.541	106.566	106.541	106.541	106.541
average fetch time (msec)	3.183	3.217	3.266	3.292	3.33	3.365
average disk utilization	0.18	0.091	0.061	0.046	0.038	0.032
Horizon 512						
fetches	8812	8812	8812	8812	8812	8812
driver time (sec)	4.406	4.406	4.406	4.406	4.406	4.406
stall time (sec)	1.444	0	0.205	0.04	0	0
elapsed time (sec)	109.388	107.944	108.149	107.984	107.944	107.944
average fetch time (msec)	3.147	3.163	3.187	3.204	3.214	3.228
average disk utilization	0.25	0.13	0.087	0.065	0.052	0.044
Horizon 1024						
fetches	8812	8812	8812	8812	8812	8812
driver time (sec)	4.406	4.406	4.406	4.406	4.406	4.406
stall time (sec)	1.533	0	0.219	0.055	0	0
elapsed time (sec)	109.477	107.944	108.163	107.999	107.944	107.944
average fetch time (msec)	3.148	3.166	3.185	3.206	3.215	3.231
average disk utilization	0.25	0.13	0.086	0.065	0.052	0.044
Horizon 2048						
fetches	8812	8812	8812	8812	8812	8812
driver time (sec)	4.406	4.406	4.406	4.406	4.406	4.406
stall time (sec)	1.533	0	0.219	0.055	0	0
elapsed time (sec)	109.477	107.944	108.163	107.999	107.944	107.944
average fetch time (msec)	3.148	3.166	3.185	3.206	3.215	3.231
average disk utilization	0.25	0.13	0.086	0.065	0.052	0.044

Table A.49: Fixed horizon performance as a function of horizon on the cscope1 trace.

Disks	1	2	3	4	5	6
Horizon 16						
fetches	4953	4953	4953	4953	4953	4953
driver time (sec)	2.4765	2.4765	2.4765	2.4765	2.4765	2.4765
stall time (sec)	3.476	0.022	0.022	0.022	0.022	0.022
elapsed time (sec)	30.887	27.433	27.433	27.433	27.433	27.433
average fetch time (msec)	3.543	3.249	3.255	3.257	3.295	3.311
average disk utilization	0.57	0.29	0.2	0.15	0.12	0.1
Horizon 32						
fetches	4953	4953	4953	4953	4953	4953
driver time (sec)	2.4765	2.4765	2.4765	2.4765	2.4765	2.4765
stall time (sec)	3.38	0.022	0.022	0.022	0.022	0.022
elapsed time (sec)	30.791	27.433	27.433	27.433	27.433	27.433
average fetch time (msec)	3.51	3.237	3.242	3.272	3.283	3.326
average disk utilization	0.56	0.29	0.2	0.15	0.12	0.1
Horizon 64						
fetches	4959	4959	4959	4959	4959	4959
driver time (sec)	2.4795	2.4795	2.4795	2.4795	2.4795	2.4795
stall time (sec)	3.121	0.012	0.012	0.012	0.012	0.012
elapsed time (sec)	30.535	27.426	27.426	27.426	27.426	27.426
average fetch time (msec)	3.524	3.251	3.277	3.301	3.333	3.368
average disk utilization	0.57	0.29	0.2	0.15	0.12	0.1
Horizon 128						
fetches	5471	5471	5471	5471	5471	5471
driver time (sec)	2.7355	2.7355	2.7355	2.7355	2.7355	2.7355
stall time (sec)	3.107	0	0	0	0	0
elapsed time (sec)	30.777	27.67	27.67	27.67	27.67	27.67
average fetch time (msec)	3.558	3.373	3.405	3.404	3.439	3.472
average disk utilization	0.63	0.33	0.22	0.17	0.14	0.11
Horizon 256						
fetches	6059	6059	6059	6059	6059	6059
driver time (sec)	3.0295	3.0295	3.0295	3.0295	3.0295	3.0295
stall time (sec)	2.726	0.135	0	0	0	0
elapsed time (sec)	30.69	28.099	27.964	27.964	27.964	27.964
average fetch time (msec)	3.624	3.398	3.427	3.424	3.464	3.504
average disk utilization	0.72	0.37	0.25	0.19	0.15	0.13
Horizon 512						
fetches	8535	8535	8535	8535	8535	8535
driver time (sec)	4.2675	4.2675	4.2675	4.2675	4.2675	4.2675
stall time (sec)	5.01	0.487	0.198	0	0	0
elapsed time (sec)	34.212	29.689	29.4	29.202	29.202	29.202
average fetch time (msec)	3.751	3.318	3.354	3.318	3.363	3.38
average disk utilization	0.94	0.48	0.32	0.24	0.2	0.16
Horizon 1024						
fetches	8535	8535	8535	8535	8535	8535
driver time (sec)	4.2675	4.2675	4.2675	4.2675	4.2675	4.2675
stall time (sec)	5.126	0.521	0.216	0	0	0
elapsed time (sec)	34.328	29.723	29.418	29.202	29.202	29.202
average fetch time (msec)	3.759	3.349	3.37	3.327	3.369	3.383
average disk utilization	0.93	0.48	0.33	0.24	0.2	0.16
Horizon 2048						
fetches	8535	8535	8535	8535	8535	8535
driver time (sec)	4.2675	4.2675	4.2675	4.2675	4.2675	4.2675
stall time (sec)	5.126	0.521	0.216	0	0	0
elapsed time (sec)	34.328	29.723	29.418	29.202	29.202	29.202
average fetch time (msec)	3.759	3.349	3.37	3.327	3.369	3.383
average disk utilization	0.93	0.48	0.33	0.24	0.2	0.16

Table A.50: Fixed horizon performance as a function of horizon on the cscope2 trace.

Disks	1	2	3	4	5	6
Horizon 16						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	37.752	28.281	21.894	16.112	13.633	11.645
elapsed time (sec)	77.844	68.373	61.986	56.204	53.725	51.737
average fetch time (msec)	10.09	16.023	17.859	18.343	18.647	19.025
average disk utilization	0.77	0.7	0.57	0.49	0.41	0.37
Horizon 32						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	35.359	25.027	17.587	12.245	8.758	6.44
elapsed time (sec)	75.451	65.119	57.679	52.337	48.85	46.532
average fetch time (msec)	9.793	15.518	17.596	18.151	18.57	18.739
average disk utilization	0.77	0.71	0.61	0.52	0.45	0.4
Horizon 64						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	32.366	21.956	14.063	8.965	5.545	3.756
elapsed time (sec)	72.458	62.048	54.155	49.057	45.637	43.848
average fetch time (msec)	9.393	14.97	17.165	18.012	18.49	18.971
average disk utilization	0.77	0.72	0.63	0.55	0.48	0.43
Horizon 128						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	30.072	19.129	11.066	6.34	3.078	1.395
elapsed time (sec)	70.164	59.221	51.158	46.432	43.17	41.487
average fetch time (msec)	9.201	14.447	16.555	17.684	18.147	18.608
average disk utilization	0.78	0.73	0.64	0.57	0.5	0.45
Horizon 256						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	25.667	14.606	7.443	4.047	1.299	0.076
elapsed time (sec)	65.759	54.698	47.535	44.139	41.391	40.168
average fetch time (msec)	8.94	13.632	15.803	17.261	17.691	18.401
average disk utilization	0.81	0.74	0.66	0.58	0.51	0.46
Horizon 512						
fetches	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	19.192	8.934	3.017	1.164	0.208	0.169
elapsed time (sec)	59.284	49.026	43.109	41.256	40.3	40.261
average fetch time (msec)	8.456	12.84	15.156	16.622	17.596	18.407
average disk utilization	0.85	0.78	0.7	0.6	0.52	0.45
Horizon 1024						
fetches	6736	6736	6736	6736	6736	6736
driver time (sec)	3.368	3.368	3.368	3.368	3.368	3.368
stall time (sec)	19.012	7.313	2.32	0.787	0.623	0.484
elapsed time (sec)	59.489	47.79	42.797	41.264	41.1	40.961
average fetch time (msec)	8.158	12.231	14.679	16.219	17.298	18.055
average disk utilization	0.92	0.86	0.77	0.66	0.57	0.49
Horizon 2048						
fetches	8299	8299	8299	8299	8299	8299
driver time (sec)	4.1495	4.1495	4.1495	4.1495	4.1495	4.1495
stall time (sec)	23.94	8.745	2.803	2.103	1.709	1.176
elapsed time (sec)	65.199	50.004	44.062	43.362	42.968	42.435
average fetch time (msec)	7.744	11.784	14.304	16.248	17.258	18.029
average disk utilization	0.99	0.98	0.9	0.78	0.67	0.59

Table A.51: Fixed horizon performance as a function of horizon on the postgres-select trace.

Disks	1	2	3	4	5	6
Horizon 16						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	36.253	16.646	9.658	7.205	5.024	3.906
elapsed time (sec)	49.273	29.666	22.678	20.225	18.044	16.926
average fetch time (msec)	15.493	15.727	15.617	15.76	15.508	15.601
average disk utilisation	0.97	0.82	0.71	0.6	0.53	0.47
Horizon 32						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	34.453	14.462	7.778	4.953	2.902	1.863
elapsed time (sec)	47.473	27.482	20.798	17.973	15.922	14.883
average fetch time (msec)	14.914	15.289	15.441	15.526	15.415	15.429
average disk utilisation	0.97	0.86	0.76	0.67	0.6	0.53
Horizon 64						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	32.46	12.642	5.718	3.059	1.206	0.51
elapsed time (sec)	45.48	25.662	18.738	16.079	14.226	13.53
average fetch time (msec)	14.407	14.853	15.033	15.11	15.217	15.386
average disk utilisation	0.98	0.89	0.83	0.72	0.66	0.58
Horizon 128						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.405	10.83	4.525	1.745	0.422	0.064
elapsed time (sec)	43.425	23.85	17.545	14.765	13.442	13.084
average fetch time (msec)	13.831	14.252	14.587	14.67	14.925	15.255
average disk utilisation	0.98	0.92	0.85	0.77	0.69	0.6
Horizon 256						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	28.591	9.283	3.336	1.104	0.183	0
elapsed time (sec)	41.611	22.303	16.356	14.124	13.203	13.02
average fetch time (msec)	13.281	13.506	13.678	14.291	14.633	15.212
average disk utilisation	0.98	0.93	0.86	0.78	0.68	0.6
Horizon 512						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	27.117	8.234	3.284	1.14	0.382	0.53
elapsed time (sec)	40.137	21.254	16.304	14.16	13.402	13.55
average fetch time (msec)	12.586	12.687	13.286	13.586	14.463	15.001
average disk utilisation	0.97	0.92	0.84	0.74	0.67	0.57
Horizon 1024						
fetches	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	23.842	6.768	2.743	1.827	1.116	1.066
elapsed time (sec)	36.862	19.788	15.763	14.847	14.136	14.086
average fetch time (msec)	11.305	11.583	12.206	13.098	13.752	13.99
average disk utilisation	0.95	0.9	0.8	0.68	0.6	0.51
Horizon 2048						
fetches	3572	3572	3572	3572	3572	3572
driver time (sec)	1.786	1.786	1.786	1.786	1.786	1.786
stall time (sec)	24.368	7.001	3.405	2.278	1.298	1.171
elapsed time (sec)	37.632	20.265	16.669	15.542	14.562	14.435
average fetch time (msec)	10.116	10.384	11.257	11.976	12.788	13.109
average disk utilisation	0.96	0.92	0.8	0.69	0.63	0.54

A.8 Performance data: *forestall* with a fixed fetch time estimate

This section contains the performance data for *forestall* with a static fetch time estimate.

Table A.52: Forestall performance as a function of static fetch time estimate on the dinero trace.

Disks	1	2	3	4	5	6
Fetch time 2						
fetches	8812	4753	4753	4753	4753	4753
driver time (sec)	4.406	2.3765	2.3765	2.3765	2.3765	2.3765
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	105.915	105.915	105.915	105.915	105.916
average fetch time (msec)	3.173	3.208	3.254	3.283	3.297	3.324
average disk utilization	0.26	0.072	0.049	0.037	0.03	0.025
Fetch time 4						
fetches	8812	10268	4909	4753	4753	4753
driver time (sec)	4.406	5.[illegible]34	2.4545	2.3765	2.3765	2.3765
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	108.672	105.993	105.915	105.915	105.916
average fetch time (msec)	3.143	3.857	3.341	3.283	3.297	3.324
average disk utilization	0.26	0.18	0.052	0.037	0.03	0.025
Fetch time 8						
fetches	8812	8818	8838	10277	4948	4852
driver time (sec)	4.406	4.409	4.419	5.1385	2.474	2.426
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	107.947	107.957	108.677	106.012	105.965
average fetch time (msec)	3.143	3.152	3.185	3.943	3.551	3.343
average disk utilization	0.26	0.13	0.087	0.093	0.033	0.026
Fetch time 15						
fetches	8812	8815	8844	8821	8830	8824
driver time (sec)	4.406	4.4075	4.422	4.4105	4.415	4.412
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	107.946	107.96	107.949	107.953	107.951
average fetch time (msec)	3.141	3.149	3.182	3.182	3.194	3.206
average disk utilization	0.26	0.13	0.087	0.065	0.052	0.044
Fetch time 30						
fetches	8812	8812	8832	8824	8816	8821
driver time (sec)	4.406	4.406	4.416	4.412	4.408	4.4105
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	107.944	107.954	107.95	107.946	107.95
average fetch time (msec)	3.142	3.147	3.177	3.182	3.19	3.204
average disk utilization	0.26	0.13	0.087	0.065	0.052	0.044
Fetch time 60						
fetches	8812	8812	8823	8816	8819	8825
driver time (sec)	4.406	4.406	4.4115	4.408	4.4095	4.4125
stall time (sec)	0.145	0	0	0	0	0.001
elapsed time (sec)	108.089	107.944	107.95	107.946	107.948	107.952
average fetch time (msec)	3.141	3.146	3.176	3.177	3.189	3.207
average disk utilization	0.26	0.13	0.087	0.065	0.052	0.044

Table A.53: Forestall performance as a function of static fetch time estimate on the cscope1 trace.

Disks	1	2	3	4	5	6
Fetch time 2						
fetches	6892	4953	4953	4953	4953	4953
driver time (sec)	3.446	2.4765	2.4765	2.4765	2.4765	2.4765
stall time (sec)	0.782	0	0	0	0	0.001
elapsed time (sec)	29.162	27.411	27.411	27.411	27.411	27.412
average fetch time (msec)	3.74	3.243	3.321	3.295	3.331	3.342
average disk utilization	0.88	0.29	0.2	0.15	0.12	0.1
Fetch time 4						
fetches	6931	8656	5108	4953	4953	4953
driver time (sec)	3.4655	4.328	2.554	2.4765	2.4765	2.4765
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.262	27.488	27.411	27.411	27.412
average fetch time (msec)	3.753	3.57	3.507	3.295	3.331	3.342
average disk utilization	0.89	0.53	0.22	0.15	0.12	0.1
Fetch time 8						
fetches	6931	8570	8680	9650	5181	5063
driver time (sec)	3.4655	4.285	4.34	4.825	2.5905	2.5315
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.219	29.274	29.759	27.525	27.467
average fetch time (msec)	3.758	3.36	3.448	3.976	3.57	3.449
average disk utilization	0.89	0.49	0.34	0.32	0.13	0.11
Fetch time 15						
fetches	6931	8570	8676	8680	8623	8582
driver time (sec)	3.4655	4.285	4.338	4.34	4.3115	4.291
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.219	29.272	29.274	29.246	29.226
average fetch time (msec)	3.759	3.362	3.438	3.368	3.394	3.359
average disk utilization	0.89	0.49	0.34	0.25	0.2	0.16
Fetch time 30						
fetches	6931	8571	8673	8688	8623	8577
driver time (sec)	3.4655	4.2855	4.3365	4.344	4.3115	4.2885
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.22	29.271	29.278	29.246	29.224
average fetch time (msec)	3.759	3.365	3.427	3.366	3.389	3.358
average disk utilization	0.89	0.49	0.34	0.25	0.2	0.16
Fetch time 60						
fetches	6931	8570	8673	8681	8627	8583
driver time (sec)	3.4655	4.285	4.3365	4.3405	4.3135	4.2915
stall time (sec)	0.911	0	0	0	0	0.001
elapsed time (sec)	29.311	29.219	29.271	29.275	29.248	29.227
average fetch time (msec)	3.758	3.362	3.43	3.363	3.394	3.36
average disk utilization	0.89	0.49	0.34	0.25	0.2	0.16

Table A.54: Forestall performance as a function of static fetch time estimate on the cscope2 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 2											
fetches	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966
driver time (sec)	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	33.592	23.879	15.09	8.553	5.27	3.226	1.792	1.258	0.812	0.262	0.018
elapsed time (sec)	73.684	63.971	55.182	48.645	45.362	43.318	41.884	41.35	40.904	40.354	40.11
average fetch time (msec)	9.476	15.459	17.353	18.156	18.318	18.707	18.866	19.181	19.048	19.212	19.31
average disk utilization	0.77	0.72	0.63	0.56	0.48	0.43	0.38	0.35	0.28	0.24	0.18
Fetch time 4											
fetches	6166	5966	5966	5966	5966	5966	5966	5966	5966	5966	5966
driver time (sec)	3.083	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	20.969	24.945	15.09	9.212	5.27	3.226	1.792	1.258	0.812	0.262	0.018
elapsed time (sec)	61.161	65.037	55.182	49.304	45.362	43.318	41.884	41.35	40.904	40.354	40.11
average fetch time (msec)	8.827	15.325	17.353	18.085	18.318	18.707	18.866	19.181	19.048	19.212	19.31
average disk utilization	0.89	0.7	0.63	0.55	0.48	0.43	0.38	0.35	0.28	0.24	0.18
Fetch time 8											
fetches	6284	6144	6025	5967	5966	5966	5966	5966	5966	5966	5966
driver time (sec)	3.142	3.072	3.0125	2.9835	2.983	2.983	2.983	2.983	2.983	2.983	2.983
stall time (sec)	15.971	14.011	20.831	11.275	7.757	3.371	1.912	1.258	0.812	0.262	0.018
elapsed time (sec)	56.222	54.192	60.953	51.368	47.849	43.463	42.004	41.35	40.904	40.354	40.11
average fetch time (msec)	8.768	13.401	14.909	17.84	18.302	18.699	18.803	19.191	19.048	19.212	19.31
average disk utilization	0.98	0.76	0.49	0.52	0.46	0.43	0.38	0.35	0.28	0.24	0.18
Fetch time 15											
fetches	6318	6333	6613	6036	5990	5969	5966	5966	5966	5966	5966
driver time (sec)	3.159	3.1665	3.3065	3.018	2.995	2.9845	2.983	2.983	2.983	2.983	2.983
stall time (sec)	15.858	5.998	2.274	6.358	5.106	1.933	2.128	0.746	0.438	0.321	0.018
elapsed time (sec)	56.126	46.274	42.69	46.485	45.21	42.027	42.22	40.838	40.53	40.413	40.11
average fetch time (msec)	8.773	13.294	14.556	16.693	17.083	18.439	18.61	19.06	19.131	19.294	19.304
average disk utilization	0.99	0.91	0.75	0.54	0.45	0.44	0.38	0.35	0.28	0.24	0.18
Fetch time 30											
fetches	6318	6592	7372	7298	7256	6664	6253	5997	5969	5970	5970
driver time (sec)	3.159	3.296	3.686	3.649	3.628	3.332	3.1265	2.9985	2.9845	2.985	2.985
stall time (sec)	15.858	5.597	1.798	0	0	0.002	0.16	0.009	0.023	0.02	0.025
elapsed time (sec)	56.126	46.002	42.593	40.758	40.737	40.443	40.396	40.117	40.117	40.114	40.119
average fetch time (msec)	8.773	13.256	14.494	16.687	16.764	17.98	18.135	18.889	19.126	19.144	19.305
average disk utilization	0.99	0.95	0.84	0.75	0.6	0.49	0.4	0.35	0.28	0.24	0.18
Fetch time 60											
fetches	6318	6592	7683	7857	8513	8152	7922	7607	7136	6672	6157
driver time (sec)	3.159	3.296	3.8415	3.9285	4.2565	4.076	3.961	3.8035	3.568	3.336	3.0785
stall time (sec)	15.858	5.597	1.798	0	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	56.126	46.002	42.749	41.038	41.366	41.186	41.07	40.922	40.682	40.446	40.188
average fetch time (msec)	8.773	13.257	14.436	16.575	16.88	17.901	18.191	18.771	19.019	19.104	19.226
average disk utilization	0.99	0.95	0.86	0.79	0.69	0.59	0.5	0.44	0.33	0.26	0.18

Table A.55: Forestall performance as a function of static fetch time estimate on the cscope3 trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 2											
fetches	11877	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739
driver time (sec)	5.9385	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	32.201	14.155	6.993	2.386	1.123	0.395	0.18	0.067	0.085	0.001	0
elapsed time (sec)	112.24	94.125	86.963	82.356	81.093	80.365	80.15	80.037	80.055	79.971	79.97
average fetch time (msec)	7.782	12.15	14.801	16.136	16.856	17.427	17.847	18.21	18.622	18.753	19.17
average disk utilization	0.82	0.76	0.67	0.58	0.49	0.42	0.37	0.33	0.27	0.23	0.18
Fetch time 4											
fetches	11989	11739	11739	11739	11739	11739	11739	11739	11739	11739	11739
driver time (sec)	5.9945	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	15.255	19.98	7.427	2.386	1.123	0.395	0.18	0.067	0.085	0.001	0
elapsed time (sec)	95.35	99.95	87.397	82.356	81.093	80.365	80.15	80.037	80.055	79.971	79.97
average fetch time (msec)	7.687	11.882	14.786	16.136	16.856	17.427	17.847	18.21	18.622	18.753	19.17
average disk utilization	0.97	0.7	0.66	0.58	0.49	0.42	0.37	0.33	0.27	0.23	0.18
Fetch time 8											
fetches	12029	12380	11935	11739	11739	11739	11739	11739	11739	11739	11739
driver time (sec)	6.0145	6.19	5.9675	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	14.198	3.272	15.339	5.365	2.089	0.439	0.18	0.067	0.085	0.001	0
elapsed time (sec)	94.313	83.563	95.407	85.335	82.059	80.409	80.15	80.037	80.055	79.971	79.97
average fetch time (msec)	7.697	11.635	13.861	16.069	16.776	17.444	17.847	18.21	18.622	18.753	19.17
average disk utilization	0.98	0.86	0.58	0.55	0.48	0.42	0.37	0.33	0.27	0.23	0.18
Fetch time 15											
fetches	12069	13014	13732	12759	12118	11739	11739	11739	11739	11739	11739
driver time (sec)	6.0345	6.507	6.866	6.3795	6.059	5.8695	5.8695	5.8695	5.8695	5.8695	5.8695
stall time (sec)	13.943	2.862	0.64	0.052	0.935	0.188	0.335	0.009	0.084	0.001	0
elapsed time (sec)	94.078	83.47	81.607	80.532	81.095	80.158	80.305	79.979	80.054	79.971	79.97
average fetch time (msec)	7.703	11.602	13.64	15.752	16.266	17.462	17.749	18.334	18.598	18.749	19.162
average disk utilization	0.99	0.9	0.77	0.62	0.49	0.43	0.37	0.34	0.27	0.23	0.18
Fetch time 30											
fetches	12092	13414	14940	14520	14134	13588	13184	12677	12078	11749	11742
driver time (sec)	6.046	6.707	7.47	7.26	7.067	6.794	6.592	6.3385	6.039	5.8745	5.871
stall time (sec)	13.943	2.862	0.64	0.052	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	94.09	83.67	82.211	81.413	81.168	80.896	80.693	80.448	80.145	79.976	79.972
average fetch time (msec)	7.741	11.586	13.625	15.76	16.23	17.39	17.76	18.613	18.804	18.924	19.105
average disk utilization	0.99	0.93	0.83	0.7	0.57	0.49	0.41	0.37	0.28	0.23	0.18
Fetch time 60											
fetches	12092	13534	15442	15702	15780	15120	14760	14393	13977	13574	12798
driver time (sec)	6.046	6.767	7.721	7.851	7.89	7.56	7.38	7.1965	6.9885	6.787	6.399
stall time (sec)	13.943	2.862	0.64	0.052	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	94.09	83.73	82.462	82.004	81.991	81.662	81.481	81.306	81.094	80.889	80.5
average fetch time (msec)	7.741	11.585	13.8	15.782	16.244	17.301	17.723	18.584	18.766	18.925	19.119
average disk utilization	0.99	0.94	0.86	0.76	0.63	0.53	0.46	0.41	0.32	0.26	0.19

Table A.56: Forestall performance as a function of static fetch time estimate on the glimpse trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 2											
fetches	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493	6493
driver time (sec)	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465	3.2465
stall time (sec)	64.134	29.651	18.465	11.895	8.142	5.704	3.725	3.291	1.773	0.821	0.507
elapsed time (sec)	106.097	71.614	60.428	53.858	50.105	47.667	45.688	45.254	43.736	42.784	42.47
average fetch time (msec)	13.314	15.231	16.228	17.452	17.993	18.334	18.457	18.528	18.598	18.642	18.707
average disk utilization	0.81	0.69	0.58	0.53	0.47	0.42	0.37	0.33	0.28	0.24	0.18
Fetch time 4											
fetches	6531	6495	6521	6493	6493	6500	6493	6493	6493	6493	6493
driver time (sec)	3.2655	3.2475	3.2605	3.2465	3.2465	3.25	3.2465	3.2465	3.2465	3.2465	3.2465
stall time (sec)	57.697	31.749	19.621	11.895	8.142	5.821	3.725	3.306	1.773	0.821	0.507
elapsed time (sec)	99.679	73.713	61.598	53.858	50.105	47.787	45.688	45.269	43.736	42.784	42.47
average fetch time (msec)	13.103	15.003	16.19	17.452	17.993	18.33	18.457	18.524	18.598	18.626	18.707
average disk utilization	0.86	0.66	0.57	0.53	0.47	0.42	0.37	0.33	0.28	0.24	0.18
Fetch time 8											
fetches	6531	6578	6538	6503	6493	6497	6493	6493	6493	6493	6493
driver time (sec)	3.2655	3.289	3.269	3.2515	3.2465	3.2485	3.2465	3.2465	3.2465	3.2465	3.2465
stall time (sec)	56.363	24.886	35.324	16.784	11.185	6.451	4.07	3.321	1.818	0.821	0.507
elapsed time (sec)	98.345	66.891	77.309	58.752	53.148	48.416	46.033	45.284	43.781	42.784	42.47
average fetch time (msec)	13.104	14.396	15.337	17.336	17.923	18.361	18.455	18.533	18.614	18.628	18.707
average disk utilization	0.87	0.71	0.43	0.48	0.44	0.41	0.37	0.33	0.28	0.24	0.18
Fetch time 15											
fetches	6531	6647	6688	6538	6530	6505	6494	6493	6493	6493	6493
driver time (sec)	3.2655	3.3235	3.344	3.269	3.265	3.2525	3.247	3.2465	3.2465	3.2465	3.2465
stall time (sec)	55.448	20.658	12.116	7.584	12.461	6.181	5.497	2.676	1.863	1.076	0.617
elapsed time (sec)	97.43	62.698	54.176	49.569	54.442	48.15	47.46	44.639	43.826	43.039	42.58
average fetch time (msec)	13.104	14.423	14.918	16.525	17.05	18.137	18.457	18.516	18.585	18.648	18.737
average disk utilization	0.88	0.76	0.61	0.54	0.41	0.41	0.36	0.34	0.28	0.23	0.18
Fetch time 30											
fetches	6565	6687	6891	6769	6723	6616	6560	6514	6493	6493	6493
driver time (sec)	3.2825	3.3435	3.4455	3.3845	3.3615	3.308	3.28	3.257	3.2465	3.2465	3.2465
stall time (sec)	55.586	19.562	6.609	2.522	1.939	0.671	0.813	0.072	0.027	0.119	0.215
elapsed time (sec)	97.585	61.622	48.771	44.623	44.017	42.695	42.809	42.045	41.99	42.082	42.178
average fetch time (msec)	13.059	14.378	14.694	16.438	16.683	17.776	17.929	18.496	18.549	18.526	18.672
average disk utilization	0.88	0.78	0.69	0.62	0.51	0.46	0.39	0.36	0.29	0.24	0.18
Fetch time 60											
fetches	6610	6687	7087	6911	6969	6823	6964	6836	6703	6624	6565
driver time (sec)	3.305	3.3435	3.5435	3.4555	3.4845	3.4115	3.482	3.418	3.3515	3.312	3.2825
stall time (sec)	54.845	19.089	5.792	2.521	1.062	0.099	0.014	0.009	0.005	0.001	0
elapsed time (sec)	96.866	61.149	48.052	44.693	43.263	42.227	42.212	42.143	42.073	42.029	41.999
average fetch time (msec)	12.998	14.374	14.565	16.422	16.75	17.762	17.634	18.282	18.499	18.486	18.626
average disk utilization	0.89	0.79	0.72	0.63	0.54	0.48	0.42	0.37	0.29	0.24	0.18

Table A.57: Forestall performance as a function of static fetch time estimate on the ld trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 2											
fetches	2900	2903	2903	2903	2903	2903	2903	2903	2903	2903	2903
driver time (sec)	1.45	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515
stall time (sec)	15.617	7.475	4.441	2.739	1.88	1.368	1.069	0.968	0.567	0.327	0.222
elapsed time (sec)	25.232	17.091	14.057	12.355	11.496	10.984	10.685	10.584	10.183	9.943	9.838
average fetch time (msec)	8.451	11.126	13.266	15.004	16.194	16.777	17.359	18.041	18.669	18.998	19.113
average disk utilization	0.97	0.94	0.91	0.88	0.82	0.74	0.67	0.62	0.53	0.46	0.35
Fetch time 4											
fetches	2981	2903	2903	2903	2903	2903	2903	2903	2903	2903	2903
driver time (sec)	1.4905	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515
stall time (sec)	15.245	8.724	4.8	2.835	1.88	1.368	1.069	0.968	0.567	0.327	0.222
elapsed time (sec)	24.9	18.34	14.416	12.451	11.496	10.984	10.685	10.584	10.183	9.943	9.838
average fetch time (msec)	8.248	10.959	13.249	15	16.194	16.777	17.359	18.041	18.669	18.998	19.113
average disk utilization	0.99	0.87	0.89	0.87	0.82	0.74	0.67	0.62	0.53	0.46	0.35
Fetch time 8											
fetches	2981	2982	3093	2903	2903	2903	2903	2903	2903	2903	2903
driver time (sec)	1.4905	1.491	1.5465	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515	1.4515
stall time (sec)	15.245	6.329	3.461	3.358	2.261	1.374	1.204	0.983	0.567	0.327	0.222
elapsed time (sec)	24.9	15.985	13.172	12.974	11.877	10.99	10.82	10.599	10.183	9.943	9.838
average fetch time (msec)	8.248	10.583	12.135	14.947	16.031	16.705	17.329	18.036	18.669	18.998	19.113
average disk utilization	0.99	0.99	0.95	0.84	0.78	0.74	0.66	0.62	0.53	0.46	0.35
Fetch time 15											
fetches	2981	2982	3137	3102	3302	2928	2909	2903	2903	2903	2903
driver time (sec)	1.4905	1.491	1.5685	1.551	1.651	1.464	1.4545	1.4515	1.4515	1.4515	1.4515
stall time (sec)	15.245	6.329	3.433	2.056	0.685	0.86	1.511	1.028	0.627	0.38	0.327
elapsed time (sec)	24.9	15.985	13.166	11.772	10.501	10.489	11.13	10.644	10.243	9.996	9.943
average fetch time (msec)	8.248	10.583	12.033	14.203	14.993	16.505	17.156	17.972	18.665	18.985	19.008
average disk utilization	0.99	0.99	0.96	0.94	0.94	0.77	0.64	0.61	0.53	0.46	0.35
Fetch time 30											
fetches	2981	2982	3137	3102	3310	3505	3737	3663	3448	3017	2917
driver time (sec)	1.4905	1.491	1.5685	1.551	1.655	1.7525	1.8685	1.8315	1.724	1.5085	1.4585
stall time (sec)	15.245	6.329	3.433	2.052	0.579	0.265	0.164	0.022	0.019	0.018	0.101
elapsed time (sec)	24.9	15.985	13.166	11.768	10.399	10.182	10.197	10.018	9.908	9.691	9.724
average fetch time (msec)	8.248	10.583	12.037	14.199	14.932	15.957	16.449	17.305	18.174	18.925	18.949
average disk utilization	0.99	0.99	0.96	0.94	0.95	0.92	0.86	0.79	0.63	0.49	0.36
Fetch time 60											
fetches	2981	2982	3137	3102	3310	3505	3734	3779	4080	4137	3880
driver time (sec)	1.4905	1.491	1.5685	1.551	1.655	1.7525	1.867	1.8895	2.04	2.0685	1.94
stall time (sec)	15.245	6.329	3.433	2.052	0.579	0.265	0.023	0.009	0.005	0.001	0
elapsed time (sec)	24.9	15.985	13.166	11.768	10.399	10.182	10.055	10.063	10.21	10.234	10.105
average fetch time (msec)	8.248	10.583	12.037	14.199	14.932	15.958	16.446	17.181	17.814	18.076	18.644
average disk utilization	0.99	0.99	0.96	0.94	0.95	0.92	0.87	0.81	0.71	0.61	0.45

Table A.58: Forestall performance as a function of static fetch time estimate on the postgres-join trace.

Disks	1	2	3	4	5	6
Fetch time 2						
fetches	3855	3856	3855	3856	3855	3856
driver time (sec)	1.9275	1.928	1.9275	1.928	1.9275	1.928
stall time (sec)	4.765	0.152	0.258	0	0	0.001
elapsed time (sec)	85.908	81.296	81.401	81.144	81.143	81.145
average fetch time (msec)	16.863	17.094	17.051	17.666	17.507	17.701
average disk utilization	0.76	0.41	0.27	0.21	0.17	0.14
Fetch time 4						
fetches	4108	3856	3855	3856	3855	3856
driver time (sec)	2.054	1.928	1.9275	1.928	1.9275	1.928
stall time (sec)	3.994	0.152	0.258	0	0	0.001
elapsed time (sec)	85.264	81.296	81.401	81.144	81.143	81.145
average fetch time (msec)	14.79	16.841	17.051	17.666	17.507	17.701
average disk utilization	0.71	0.4	0.27	0.21	0.17	0.14
Fetch time 8						
fetches	4695	4174	3937	3856	3855	3856
driver time (sec)	2.3475	2.087	1.9685	1.928	1.9275	1.928
stall time (sec)	3.995	0.152	0.293	0	0	0.001
elapsed time (sec)	85.558	81.455	81.477	81.144	81.143	81.145
average fetch time (msec)	15.015	14.987	15.805	17.482	17.477	17.701
average disk utilization	0.82	0.38	0.25	0.21	0.17	0.14
Fetch time 15						
fetches	4698	5803	6051	4044	3922	3872
driver time (sec)	2.349	2.9015	3.0255	2.022	1.961	1.936
stall time (sec)	3.994	0.153	0.258	0	0	0.001
elapsed time (sec)	85.559	82.27	82.499	81.238	81.177	81.153
average fetch time (msec)	15.033	16.514	15.716	16.161	16.335	17.349
average disk utilization	0.83	0.58	0.38	0.2	0.16	0.14
Fetch time 30						
fetches	4698	5833	6194	6127	6200	5032
driver time (sec)	2.349	2.9165	3.097	3.0635	3.1	2.516
stall time (sec)	3.994	0.153	0.258	0	0	0.001
elapsed time (sec)	85.559	82.285	82.571	82.279	82.316	81.733
average fetch time (msec)	15.032	16.567	15.956	16.585	16.583	17.107
average disk utilization	0.83	0.59	0.4	0.31	0.25	0.18
Fetch time 60						
fetches	4698	5837	6224	6160	6042	5910
driver time (sec)	2.349	2.9185	3.112	3.08	3.021	2.955
stall time (sec)	3.994	0.153	0.258	0	0	0.001
elapsed time (sec)	85.559	82.287	82.586	82.296	82.237	82.172
average fetch time (msec)	15.03	16.577	15.937	16.598	16.713	17.145
average disk utilization	0.83	0.59	0.4	0.31	0.25	0.21

Table A.59: Forestall performance as a function of static fetch time estimate on the postgres-select trace.

Disks	1	2	3	4	5	6	7	8	10	12	16
Fetch time 2											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.691	11.831	4.806	1.562	0.524	0.143	0.032	0.009	0.005	0.001	0
elapsed time (sec)	43.711	24.851	17.826	14.582	13.544	13.163	13.052	13.029	13.025	13.021	13.02
average fetch time (msec)	13.985	14.783	14.672	15.051	14.995	15.409	15.265	15.549	15.372	15.21	15.114
average disk utilization	0.99	0.92	0.85	0.8	0.68	0.6	0.52	0.46	0.36	0.3	0.22
Fetch time 4											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.691	10.837	5.617	1.562	0.524	0.143	0.032	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.857	18.637	14.582	13.544	13.163	13.052	13.029	13.025	13.021	13.02
average fetch time (msec)	13.985	14.13	14.653	15.051	14.995	15.409	15.265	15.549	15.372	15.21	15.114
average disk utilization	0.99	0.91	0.81	0.8	0.68	0.6	0.52	0.46	0.36	0.3	0.22
Fetch time 8											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.691	10.811	3.517	0.868	0.736	0.08	0.032	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.831	16.537	13.888	13.756	13.1	13.052	13.029	13.025	13.021	13.02
average fetch time (msec)	13.985	14.164	13.915	14.467	14.835	15.339	15.267	15.549	15.372	15.21	15.114
average disk utilization	0.99	0.92	0.87	0.8	0.67	0.6	0.52	0.46	0.36	0.3	0.22
Fetch time 15											
fetches	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425	1.5425
stall time (sec)	30.691	10.791	3.517	0.844	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.811	16.537	13.864	13.02	13.021	13.02	13.029	13.025	13.021	13.02
average fetch time (msec)	13.985	14.166	13.936	14.514	14.38	15.096	14.868	15.383	15.289	15.182	15.132
average disk utilization	0.99	0.92	0.87	0.81	0.68	0.6	0.5	0.46	0.36	0.3	0.22
Fetch time 30											
fetches	3085	3085	3085	3085	3085	3297	3764	3588	3123	3085	3085
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.5425	1.6485	1.882	1.794	1.5615	1.5425	1.5425
stall time (sec)	30.691	10.772	3.517	0.844	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.792	16.537	13.864	13.02	13.127	13.36	13.281	13.044	13.021	13.02
average fetch time (msec)	13.985	14.172	13.947	14.536	14.398	15.058	15.214	15.234	15.241	15.096	15.052
average disk utilization	0.99	0.92	0.87	0.81	0.68	0.63	0.61	0.51	0.36	0.3	0.22
Fetch time 60											
fetches	3085	3085	3085	3085	3166	3313	3830	3942	3904	3860	4108
driver time (sec)	1.5425	1.5425	1.5425	1.5425	1.583	1.6565	1.915	1.971	1.952	1.93	2.054
stall time (sec)	30.691	10.772	3.517	0.844	0	0.001	0	0.009	0.005	0.001	0
elapsed time (sec)	43.711	23.792	16.537	13.864	13.061	13.135	13.393	13.458	13.435	13.409	13.532
average fetch time (msec)	13.985	14.173	13.95	14.548	14.243	15.037	15.228	15.284	14.8	14.764	14.865
average disk utilization	0.99	0.92	0.87	0.81	0.69	0.63	0.62	0.56	0.43	0.35	0.28

Table A.60: Forestall performance as a function of static fetch time estimate on the xds trace.

Disks	1	2	3	4	5	6
Fetch time 2						
fetches	5925	5912	5889	5891	5889	5897
driver time (sec)	2.9625	2.956	2.9445	2.9455	2.9445	2.9485
stall time (sec)	30.831	3.009	3.435	1.188	0.52	0.1
elapsed time (sec)	63.872	36.044	36.458	34.212	33.543	33.127
average fetch time (msec)	10.717	7.708	14.253	10.062	15.601	11.066
average disk utilization	0.99	0.63	0.77	0.43	0.55	0.33
Fetch time 4						
fetches	5925	6016	5907	5894	5890	5897
driver time (sec)	2.9625	3.008	2.9535	2.947	2.945	2.9485
stall time (sec)	30.667	0.421	3.216	0.707	0.546	0.1
elapsed time (sec)	63.708	33.508	36.248	33.733	33.57	33.127
average fetch time (msec)	10.711	7.706	14.132	9.903	15.673	11.06
average disk utilization	1	0.69	0.77	0.43	0.55	0.33
Fetch time 8						
fetches	5925	7045	6444	6025	5910	5896
driver time (sec)	2.9625	3.5225	3.222	3.0125	2.955	2.948
stall time (sec)	30.667	0.337	0.355	0.16	0.225	0.06
elapsed time (sec)	63.708	33.938	33.656	33.251	33.259	33.087
average fetch time (msec)	10.711	7.499	14.085	10.019	14.989	10.639
average disk utilization	1	0.78	0.9	0.45	0.53	0.32
Fetch time 15						
fetches	5925	7274	6563	8643	7534	6421
driver time (sec)	2.9625	3.637	3.2815	4.3215	3.767	3.2105
stall time (sec)	30.667	0.337	0.356	0.129	0.133	0.055
elapsed time (sec)	63.708	34.053	33.716	34.529	33.979	33.344
average fetch time (msec)	10.711	7.492	14.095	9.717	15.34	10.458
average disk utilization	1	0.8	0.91	0.61	0.68	0.34
Fetch time 30						
fetches	5925	7742	6563	9699	8300	9846
driver time (sec)	2.9625	3.871	3.2815	4.8495	4.15	4.923
stall time (sec)	30.667	0.337	0.356	0.129	0.133	0.055
elapsed time (sec)	63.708	34.287	33.716	35.057	34.362	35.057
average fetch time (msec)	10.711	7.467	14.099	9.758	15.445	10.708
average disk utilization	1	0.84	0.91	0.67	0.75	0.5
Fetch time 60						
fetches	5925	7778	6563	9807	8300	10015
driver time (sec)	2.9625	3.889	3.2815	4.9035	4.15	5.0075
stall time (sec)	30.667	0.337	0.356	0.129	0.133	0.055
elapsed time (sec)	63.708	34.305	33.716	35.111	34.362	35.141
average fetch time (msec)	10.711	7.497	14.096	9.798	15.451	10.678
average disk utilization	1	0.85	0.91	0.68	0.75	0.51

Bibliography

[1] H.M. Abdel-Wahab and T. Kameda. Scheduling to Minimize Maximum Cumulative Cost Subject to Series-Parallel Precedence Constraints. *Operations Research*, 26(1):141-158, 1978.

[2] 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.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison–Wesley, Reading, MA, 1983, pp. 135–145.

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

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

[6] 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.

[7] 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.

[8] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. *ACM Transactions on Computer Systems (TOCS)* 14(4):311-343, November 1996.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979, pp. 199-200.

[15] Garth A. Gibson. Personal communication.

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

[17] 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.

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

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1996.

[20] 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.

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

[22] Tracy Kimbrel, Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Integrated Parallel Prefetching and Caching. Princeton University Computer Science Department Tech Report TR-502-95, November 1995. Short version in *1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

[23] 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.

[24] 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. In *Proceedings of*

the ACM SIGOPS/USENIX Association Symposium on Operating System Design and Implementation (OSDI), October 1996.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-inserted I/O Prefetching for Out-of-core Applications. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, October 1996, pp. 3–17.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread Scheduling for Cache Locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60-71, October, 1996.

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

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

[40] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts, Fourth Edition.* Addison–Wesley, Reading, MA, 1994, pp. 312–325.

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

[42] 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.

[43] Andrew Tomkins, R. Hugo Patterson, and Garth A. Gibson. Informed Multi-process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

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

[45] John Wilkes. Personal communication.

Vita

Tracy Kimbrel was born in Wichita, Kansas on January 25, 1961. He received the Bachelor of Science degree in Computer Science with highest distinction, from the University of Washington in 1986. He worked at The Boeing Company from 1986 until 1992. He entered the graduate school at the University of Washington in Seattle in 1989. Tracy received the Master of Science degree in Computer Science and Engineering in 1991. He completed the Ph.D. in Computer Science and Engineering in 1997.