Reducing Seek Overhead with Application-Directed Prefetching

Steve VanDeBogart Christopher Frost Eddie Kohler
UCLA
http://libprefetch.cs.ucla.edu/

Abstract

An analysis of performance characteristics of modern disks finds that prefetching can improve the performance of nonsequential read access patterns by an order of magnitude or more, far more than demonstrated by prior work. Using this analysis, we design prefetching algorithms that make effective use of primary memory, and can sometimes gain additional speedups by reading unneeded data. We show when additional prefetching memory is most critical for performance. A contention controller automatically adjusts prefetching memory usage, preserving the benefits of prefetching while sharing available memory with other applications. When implemented in a library with some kernel changes, our prefetching system improves performance for some workloads of the GIMP image manipulation program and the SQLite database by factors of 4.9x to 20x.

1 Introduction

Modern magnetic disks are, as is well known, dramatically slower at random reads than sequential reads. Technological progress has exacerbated the problem; disk throughput has increased by a factor of 60 to 85 over the past twenty-five years, but seek times have decreased by a factor of only 15.[1] Disks are less and less like random-access devices in terms of performance. Although flash memory reduces the cost differential of random accesses (at least for reads—many current SSD disks have terrible performance on small random writes), disks continue to offer vast amounts of inexpensive storage. For the foreseeable future, it will remain important to optimize the performance of applications that access disk-like devices—that is, devices with much faster sequential than random access.

Operating systems heavily optimize their use of disks, minimizing the volume of transferred data while opportunistically striving to make requests sequential. Large buffer caches ensure that disk reads are only done when necessary, write buffering helps to batch and minimize writes, disk scheduling reorders disk requests to group them and minimize seek distances, and readahead expands small read requests into more efficient large requests by predicting applications' future behavior. These techniques apply well to workloads whose accesses are already sequential or near-sequential, for which they achieve performance near hardware capabilities. For nonsequential access patterns, however, the techniques break down. Readahead implementations, for example, often turn off after detecting such patterns: future nonsequential accesses are by nature hard to predict, making it more likely that prediction mistakes would pollute the buffer cache with irrelevant data. Unfortunately, though careful design of application on-disk data structures can make common-case accesses sequential, many applications must sometimes access data nonsequentially—for instance, to traverse a giant database by a non-primary index—and any nonsequential access pattern is radically slow.

The best solution to this performance problem is to avoid critical-path disk access altogether, such as by obtaining enough memory to hold all application data. Failing that, distributing data over several disks or machines can reduce the overall cost of random access by performing seeks in parallel [18]. However, these solutions may not always apply—even resource-constrained users can have large data sets—and any technique that speeds up single-disk nonsequential accesses is likely to improve the performance of distributed solutions.

We present an *application-directed prefetching* system that speeds up application performance on single-disk nonsequential reads by, in some cases, more than an order of magnitude.

In application-directed prefetching systems, the application informs the storage system of its intended upcoming reads. (Databases, scientific workloads, and others are easily able to calculate future accesses [13, 17].) Previous work on application-directed caching and prefetching demonstrated relatively low speedups for single-process, single-disk workloads (average speedup 26%, maximum 49%) [2, 18]. However, this work aimed to overlap CPU time and I/O fetch time without greatly increasing memory use, and thus prefetched relatively little data from disk (16 blocks) just before a process needed it. Our system, libprefetch, aims solely to minimize I/O fetch time, a better choice given today's widened gap between processor and disk performance. The prefetching system is aggressive, fetching as much data as fits in available memory. It is also relatively simple, fitting in well with existing operating system techniques; most code is in a user-space library. Small, but critical, changes in kernel behavior help ensure that prefetched data is kept until it is used. A contention controller detects changes in available memory and compensates by resizing the prefetching window, avoiding performance collapse when prefetching applications compete for memory and increasing performance when more memory is available. Our measurements show substantial speedups on test workloads, such as a 20x speedup on a SQLite table scan of a data set that is twice the size of memory. Running concurrent instances of applications with libprefetch shows similar factors of improvement.

Our contributions include our motivating analysis of seek time; our prefetching algorithm; the libprefetch interface, which simplifies applications' access to prefetching; the contention controller that prevents libprefetch from monopolizing memory; and our evaluation. Section 2 describes related work, after which Section 3 uses disk benchmarks to systematically build up our prefetching algorithm. Section 4 describes the libprefetch interface and its implementation. Finally, Sections 5 and 6 evaluate libprefetch's performance and conclude.

2 Related Work

Fueled by the long-growing performance gulf between disk and CPU speeds, considerable research effort has been invested in improving disk read performance by caching and prefetching. Prefetching work in particular has been based on predicted, application-directed, and inferred disk access patterns.

Disk Modeling Ruemmler and Wilkes [19] is the classic paper on disk performance modeling. Our seek time observations complement those of Schlosser et al. [20]; like them, we use our observations to construct more effective ways to use a disk.

Predicting Accesses Operating systems have long employed predictive readahead algorithms to speed up sequential file access. This improves performance for many workloads, but can retard performance if future accesses are mispredicted. As a result, readahead algorithms usually don't try to improve less predictable access patterns, such as sequential reads of many small files or nonsequential reads of large files.

Dynamic history-based approaches [5, 8, 12, 14–16, 23, 27] infer access patterns from historical analysis, and so are not limited to simple patterns. However, requiring historical knowledge has limitations: performance is not improved until a sufficient learning period has elapsed, non-repetitive accesses are not improved at all, and the historical analysis can impose significant memory and processing overheads.

Application-Directed Accesses Cao et al. [2] and Patterson et al. [18] present systems like libprefetch where applications convey their access patterns to the file system to increase disk read performance. While all three systems prefetch data to reduce application runtime, the past decade's hardware progress has changed the basic disk performance bottlenecks. Whereas prior systems prefetch to hide disk latency by overlapping CPU time and I/O time, libprefetch prefetches data to minimize I/O time and increase disk throughput by reducing seek distances. This approach permits much greater performance increases on today's computers. Specifically, the designs of Cao et al.'s ACFS and Patterson et al.'s system are based on the simplifying assumption that block fetch time is fixed, independent of both the block's location relative to the disk head and the block's absolute location on disk. Based on this disk model, their derived optimal prefetching rules state that 1) blocks must be prefetched from disk in precisely application access order, and 2) a block must be prefetched as soon as there is available RAM. Because RAM was so scarce in systems of the time, this had the effect of retrieving data from disk just before it was needed, although Cao et al. also note that in practice request reordering can provide a significant performance improvement for the disk. Their implementation consists of one piece that takes great care to prefetch the very next block as soon as there is memory, and a second piece that buffers 4 to 16 of these requests to capitalize on disk ordering benefits. This approach is no longer the best trade-off; increased RAM sizes and larger performance gaps between disks and processors make it more important to maximize disk throughput. Libprefetch thus actively waits to request disk data until it can prefetch enough blocks to fill a significant portion of memory. For seek-limited applications on today's systems, minimizing seek distances by reordering large numbers of blocks reduces I/O time and application runtime significantly more than prior approaches.

Both Patterson's system and Cao's ACFS explicitly addressed process coordination, especially important since their implementations replaced the operating system's cache eviction policy. Libprefetch, in contrast, implements prefetching in terms of existing operating system mechanisms, so a less coordinated approach suffices. Existing operating system algorithms balance I/O among multiple processes, while libprefetch's internal contention controller automatically detects and adapts to changes in available memory. In this regard, our approaches are complementary; something like ACFS's two-level caching might further improve prefetching performance relative to Linux's default policy.

Using a modest amount of RAM to cache prefetches, Patterson et al. and Cao et al. achieved maximum single-disk improvements of 55% (2.2x) and 49% (2x), respectively. Patterson et al. used multiple disks to achieve additional speedups, whereas libprefetch uses additional

RAM to achieve speedups of as much as 20x. For concurrent process performance Patterson et al. report a maximum performance improvement of 65% (2.9x), Cao et al. 76% (4.2x); we see improvements of 4x to 23x.

Inferred Accesses Rather than requiring the application to explicitly supply a list of future reads, a prefetching system can automatically generate the list—either from application source code, using static analysis [1, 4, 25, 26], or from the running application, using speculative execution [3, 7]. Static analysis can generate file read lists, but data dependence and analytic imprecision may limit these methods to simple constructs that do not involve abstractions over I/O. Speculative-execution prefetchers use spare CPU time to tell the operating system what file data will be needed. Speculation can provide benefits for unmodified applications, and is especially useful when it is difficult to programmatically produce the access pattern. Libprefetch could serve as the back end for a system that determined access patterns using analysis or speculation, but the less-precise information these methods obtain might reduce prefetching's effectiveness relative to our results.

Libprefetch's performance benefits are competitive with these other systems. For example, Chang et al. [3] focus on parallel disk I/O systems that provide more I/O bandwidth than is used by an unmodified application. Libprefetch obtains more prefetching benefit with one disk than Chang et al. find going to four.

Prefetching in Databases Lacking good OS support [21], applications like databases have long resorted to raw disk partitions to, for example, implement their own prefetch systems [22]. Better OS mechanisms, such as libprefetch, may reduce the need for this duplication of effort.

POSIX Asynchronous I/O The POSIX Asynchronous I/O [10], **posix_fadvise**, and **posix_madvise** [9] interfaces allow applications to request prefetching, but current implementations of these interfaces tend to treat prefetching advice as mandatory and immediate. To achieve good performance, applications must decide when to request a prefetch, how much to prefetch, and how to order requests. Libprefetch uses **posix_fadvise** as part of its implementation and manages these details internally.

3 The Impact of Modern Disk Characteristics on Prefetching

Disk prefetching algorithms aim to improve the performance of future disk reads by reading data before it is needed. Since our prefetching algorithm will use precise application information about future accesses, we need not worry about detecting access patterns or trying to predict what the application will use next. Instead, the chief goal is to determine the fastest method to retrieve the requested data from disk.

This section uses disk benchmarks to systematically build up a prefetching algorithm that takes advantage of the strengths, and as much as possible avoids the weaknesses, of modern I/O subsystems. The prefetching algorithm must read at least the blocks needed by the application, so there are only a few degrees of freedom available. The prefetcher can reorder disk requests within the window of memory available for buffering, it can combine disk requests, and it can read non-required data if that would help. Different disk layout or block allocation algorithms could also lead to better performance, but these file system design techniques are orthogonal to the issues we consider.

3.1 Seek Performance

Conventional disk scheduling algorithms do not know what additional requests will arrive in the future, leading to a relatively small buffer of requests that can be reordered. In contrast, a prefetching algorithm that does know future accesses can use a reorder buffer as large as available memory. A larger buffer can substantially reduce average seek distance. In this section, we measure the actual cost of various seek distances on modern disks, aiming to determine where seek distance matters and by how much.

We measured the average time to seek various distances, both forward and backward. Because the seek operation is below the disk interface abstraction, it is only possible to measure a seek in conjunction with a read or write operation. Therefore, the benchmarks start by reading the first block of the disk to establish the disk head location (or the last block, if seeking backward), then read several blocks from the disk, each separated by the seek distance being tested. With this test methodology, a seek distance of zero means that we read sequential disk locations with multiple requests, and a seek distance of -1 block re-reads the same block repeatedly. All the tests in this section use Direct I/O to skip the buffer cache, ensuring that buffer cache hits do not optimize away the effects we are trying to measure. See the evaluation section for a description of our experimental setup.

The results of running this benchmark on two disks is shown in Figure 1. The "Read" lines in the graphs, included for comparison, show the time to read the given amount of data, assuming maximum throughput. In either direction, the cost of large ($\geq$1MB) seeks is substantial, ranging from 5ms to 10ms; avoiding just 200 of these seeks could reduce runtime by one to two seconds. The cost is similar in either direction, except that seeks with distance less than 1MB are faster when seeking forward than seeking backward. This motivates the choice

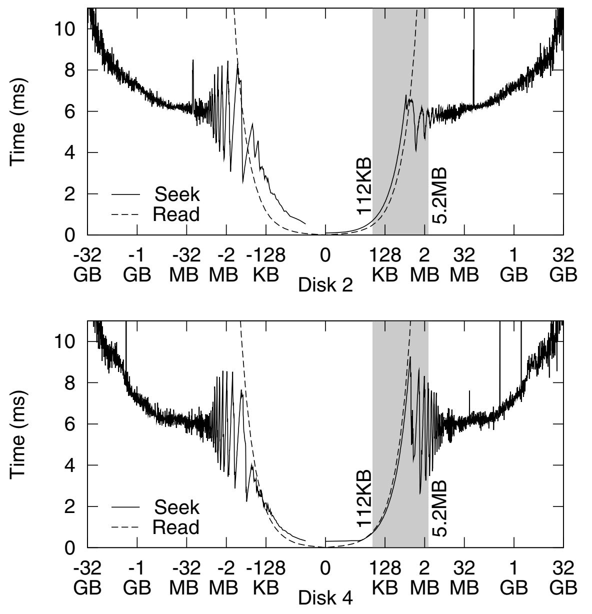


Figure 1: Average time to seek a given distance (forward or backward) compared to maximum read throughput. The oscillations are due to disk geometry combined with rotational latency. The grey region highlights where seek time changes most dramatically.

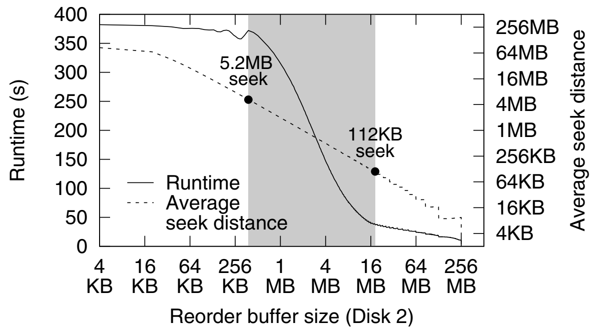


Figure 2: Effect of reorder buffer size on runtime and average seek distance for random reads. The test reads a total of 256MB of data. Along the x axis, we vary the reorder buffer size (the amount of data prefetched at once). Within each buffer, disk requests are sorted by logical block number. Runtime changes most dramatically in the grey region, where average seek distance drops from 5.2MB to 112KB, the boundaries of the highlighted region in Figure 1.

of a Circular LOOK algorithm: scan forward servicing requests until there are no requests past the head position, then return to the request with the lowest offset and repeat.

Seek time increases by roughly a factor of five from around 112KB to roughly 1 to 5MB, shown as the highlighted regions in the graphs. In contrast, seek times for distances above 5MB increase slowly (note the graph's log-scale x axis), by about a factor of two. Not considering the disk geometry effects visible as oscillations, a disk scheduling algorithm should minimize seek distance; though not all seek reductions are equal, reducing medium seeks far below 1MB will have more impact than reducing very large seeks to 1MB or more.

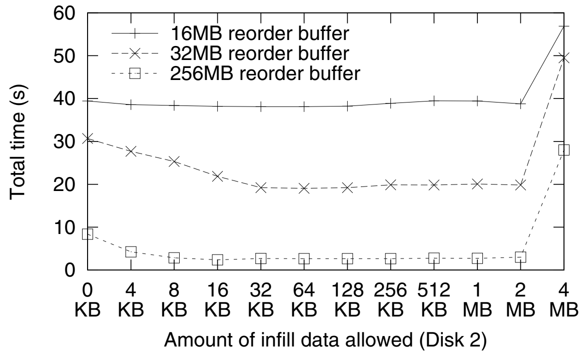
Figure 1 also shows the unexpected result that for distances up to 32KB, it may be cheaper to read that amount of data than to seek. This suggests that adjacent requests with small gaps might be serviced faster by requesting the entire range of data and discarding the uninteresting data. Our prefetching algorithm implements this feature, which we call *infill*.

3.2 Effect of Reorder Buffer Size

To reorder disk requests, the prefetch system must buffer data returned ahead of the application's needs. Thus, the window in which the prefetch system can reorder requests is proportional to the amount of memory it can use to store their results. So how much memory should be used to reorder prefetch data? Is there a threshold where more memory doesn't improve runtime substantially? The next benchmark tries to answer these questions by using various reorder buffer sizes. The benchmark is an artificial workload of 256MB of randomly chosen accesses to a 256MB file; some blocks in the file may be accessed multiple times and others may not be accessed at all, but the test uses Direct I/O so all the requests are satisfied from disk. Benchmarks for different amounts of total data had similar results.

The benchmark proceeds through the 256MB workload one reorder buffer at a time, reading pages within each buffer in C-LOOK order (that is, by increasing disk position). Figure 2 shows how the size of the reorder buffer affects runtime. The region of the graph between reorder buffer sizes of 384KB and 18MB, highlighted in grey, shows the most dramatic change in runtime. Examining the average seek distance of the resulting accesses helps to explain the change within the grey region. As the reorder buffer grows, the average seek distance in the grey region decreases from 5.2MB to 112KB (dotted line, right-hand y axis). This range of seek distances is greyed in Figure 1 and corresponds to the region where seek cost changes most dramatically.

However, increasing the reorder buffer beyond 18MB still has a substantial effect, decreasing runtime from 37.6 seconds to 8.4 seconds, an additional speedup of 4.7x. This continued decrease in runtime is due in part to the reduction in the number of disk passes needed to retrieve the data, from about 14 with a reorder buffer of 18MB down to one with a reorder buffer of 256MB. This demonstrates that the prefetching algorithm should prefetch as much as possible, but at least enough to reduce the average seek distance to 112KB, if possible.



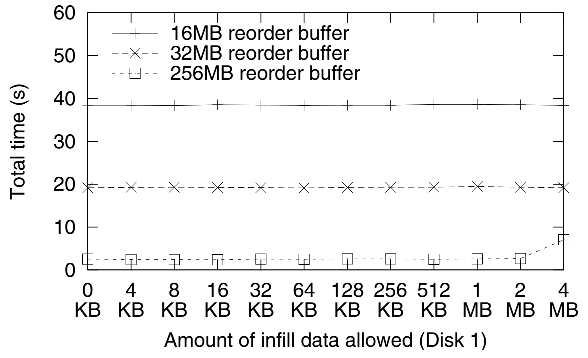


Figure 3: Effect of infill on runtime. The test reads a total of 256MB of data. Along the x axis, we vary the maximum amount of extra data read between requests. Infill of 32KB maximizes performance improvement; infill beyond 2MB can hurt performance.

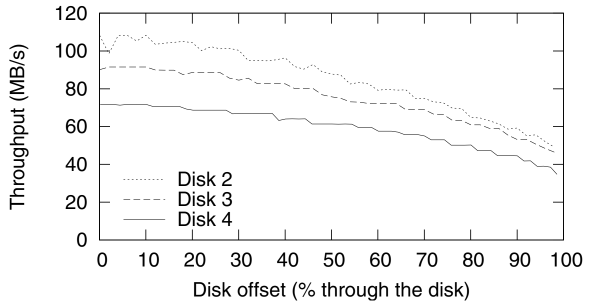


Figure 4: Read throughput at various disk offsets. This test reads 256MB at each sampled offset.

3.3 Effect of Infill

Figure 1 suggested that reading and discarding small gaps between requests might be faster than seeking over those gaps. We modified the previous benchmark to add infill, varying the maximum infill allowed. Figure 3 shows that infill of up to 32KB reduces runtime on Disk 2. This corresponds to the region of Figure 1 where seeks take longer than similarly-sized maximum-throughput reads. Infill amounts between 32KB and 2MB have no additional effect, corresponding to the region of the seek graph where seek time is equal to read time for an equivalent amount of data. When infill is allowed to exceed 2MB, runtime increases, confirming that reads of this size are more expensive than seeking. For the 16MB reorder buffer dataset, infill has very little effect: as Figure 2 shows, the average seek distance for this test is 128KB, above the threshold where we expect infill to help. On Disk 1, however, infill was performance-neutral. Apparently its firmware makes infill largely redundant.

In summary, our tests show that infill amounts up to 32KB can help performance on some disks, but only for datasets that have a significant number of seeks less than 32KB in size.

3.4 Other Techniques

Conventional wisdom says that read throughput is better at the outer edge of disk platters (low logical block numbers), which we confirm on several disks in Figure 4. The data for Disk 4 clearly shows the transitions between different areal densities (where the line steps down). The global effect is significant, slowing by about 50% from the first to the last logical block number. However, with large disks, even large files will usually span a small fraction of the disk, leading to a small speed difference between the beginning and end of a file.

It is also conventional wisdom that larger disk requests achieve better throughput. We tested this by reading a large amount of data (128MB) with a number of differently-sized read requests. Initially, this benchmark indicated that the size of read requests did not have an impact on performance, although performance differed when Linux's readahead was enabled. Upon examining block traces, we discovered that the request size issued to the disk remained constant, even as the application issued larger reads. With readahead enabled, large disk requests were always issued; with readahead disabled, disk requests were always 4KB in size. (It is odd that disabling readahead would cause Linux to always issue 4KB disk requests, even when the application requests much more data in a single read call; we call this a bug.) Using Direct I/O caused Linux to adhere to the request size we issued. Figure 5 shows that request sizes below 64KB do not achieve maximum throughput, although Linux's readahead code normally compensates for this.

3.5 Prefetching Algorithm

In summary, large reorder buffers lead to significant speed improvements, so prefetching should use as much memory as is available. Forward seeks can be faster than backward seeks, so prefetching should use a C-LOOK style algorithm for disk scheduling. Small amounts of infill can be faster than seeking between requests on some disks and have no negative effect on others, so infill should be used. The request size can have a significant impact on performance, but the I/O subsystem usually

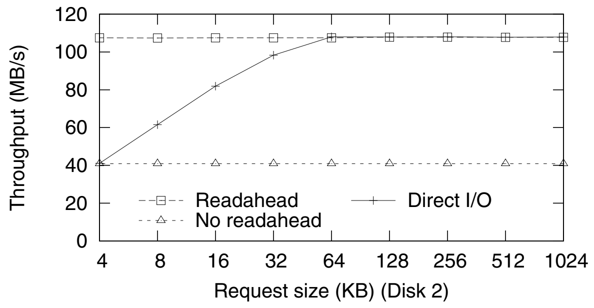


Figure 5: Effect of request size on throughput. Using or disabling readahead shows the two extremes of the performance spectrum. Direct I/O shows the true effect of request size on throughput.

optimizes this parameter sufficiently. Thus, libprefetch's basic prefetching algorithm simply traverses an access list one reorder buffer at a time, prefetching each buffer's data in disk order. Although further optimizations might be possible—for example, for some access patterns it may be more advantageous to prefetch a partial reorder buffer—we currently implement the basic version.

The tests also help us understand how prefetching can scale. The reorder buffer improves performance by reducing average seek distance, but not all reductions are equally valuable. For example, Figure 1 indicates that reducing average seek distance 500x from 1GB to 2MB wouldn't improve performance much. The amount a given reorder buffer reduces average seek distance depends on access pattern and file system layout, but for uniform random accesses and sequential layout, we can approximate the resulting seek distance analytically. Given an N-page sequential file and a reorder buffer of K pages ($2 \leq K \ll N$), the expected average seek distance will be $2N(K-1)/K(K+1)$ pages.[2] This distance is roughly proportional to the ratio of file size to reorder buffer size. Thus, increasing memory by some factor will reduce average seek distance by the same factor, or, equivalently, produce the same seek distance when processing a proportionally bigger file. The formula can also be used to predict when seek distance will rise into the unproductive region of Figure 2. For instance, assuming 3GB of RAM available for prefetching and uniform random accesses, a 384GB file will achieve an average seek distance of 1MB, the rough "large seek" boundary.

4 Libprefetch

The libprefetch library implements our prefetching algorithm underneath a callback-based interface that easily integrates into applications. Libprefetch calls an application's callback when it needs more prefetching information. The callback can compute the application's list of future accesses and pass it to libprefetch's **request_prefetching** function. The computed list can replace or augment the current access list.

```
struct access_entry {              #define PF_APPEND          1
    loff_t pageOffset;             #define PF_SET             2
    int fd;                        #define PF_SET_FROM_MARK   4
    bool mark;                     #define PF_DONE            8
};

typedef void (*callback_t)(void * arg,
                           int lastMarkedFD, loff_t lastMarkedOffset,
                           int requestedFD, loff_t requestedOffset);
ssize_t request_prefetching(client_t c, const struct access_entry * a,
                            size_t n, int type);

client_t register_client(callback_t cb, void * arg);
int unregister_client(client_t c);

region_t register_region(client_t c, int fd, loff_t start, loff_t end);
int unregister_region(client_t c, region_t r);

int ignore_accesses(client_t c);
int unignore_accesses(client_t c);
```

Figure 6: Libprefetch interface. Applications create a client with **register_client**, then declare regions of files where accesses will be prefetched using **register_region**. When libprefetch needs an updated access list it calls back into the application with the registered callback function. This callback updates the access list with calls to **request_prefetching**. Finally, **ignore_accesses** lets the application read data from a prefetchable region without affecting the access list.

Libprefetch periodically asks the kernel to prefetch a portion of the access list; how much to prefetch depends on available memory. As the access list is consumed, or if actual accesses diverge from it, libprefetch calls back into the application to extend the list. Libprefetch tracks the application's progress through the list by overriding the C library's implementations of **read**, **readv**, and **pread**.

Figure 6 summarizes the libprefetch interface. The rest of this section discusses libprefetch in more detail, including a design rationale and important aspects of its implementation.

4.1 Callbacks

Libprefetch's callback-based design achieves the following goals:

- The interface should minimize interference with application logic.
- The application should not have to guess when to make new prefetch requests. Therefore, libprefetch should actively request new prefetch information from the application. Lower-level components like libprefetch or the system's buffer cache manager best know when prior prefetching results have completed, indicating the need for additional prefetching, or when a read request blocks, indicating that the application's access list was inaccurate.

This model lets applications isolate most access-list management logic in a self-contained callback function. Of course, an application that prefers to actively manage the access list can do so.

Libprefetch issues a callback from its implementation of **read**, **readv**, or **pread**. The callback is passed several arguments indicating the application's position in the access list. (This further isolates prefetching from application logic, since the application need not explicitly track this position.) These arguments include a user-specified **void ***, the file descriptor and offset of the access that triggered the callback, and the file descriptor and offset of the most recently accessed *marked page*. Marked pages are application-specified access-list landmarks that can be more useful to the callback than the current position. For example, consider a database accessing data via an index. The index's pages are accessed once each in a predictable, often sequential, order, but the data pages may be accessed seemingly randomly (and multiple times each, if the data set doesn't fit in memory). This makes libprefetch's position in the index portion of the access list more useful for planning purposes than its position in the data page portion. The database thus marks index pages within its access list, allowing its callback to quickly determine the most recently read index page, and therefore how far reading has progressed. Note that the most recent marked page need not have actually been read, as long as subsequent pages were read. This can happen, for example, when an index page was already in an application-level cache. Libprefetch correctly handles such small divergences from a predicted access list.

In addition to providing a means to specify the callback function, the **register_client** interface would let multiple threads within an application specify their own access lists. Our current implementation of libprefetch does not support multiple clients per process, though we expect this feature would be easy to implement for clients with non-overlapping file regions.

4.2 The Access List

The application interface for specifying future accesses was designed to achieve three goals:

- The basic interface for requesting prefetching should be the simplest sensible interface that can represent arbitrary access patterns, such as a list of future accesses in access order.
- Prefetching should work both within and across files.
- The application should be able to define its access pattern incrementally. The data structures required to specify a large pattern would take up memory that could otherwise be used for data. More fundamentally, some applications only gradually discover their access patterns.

The application specifies its access list by filling in an array of **access_entry** structures with the file descriptor and offset for each intended access. An arbitrary subset of these structures can be marked. If the application will access block *A*, then block *B*, and then block *A* again, it simply adds those three entries to the array. It passes this array to libprefetch's **request_prefetching** function. Each call to **request_prefetching** can either replace the current list, append to the list, or replace the portion of the current list following the most-recently-accessed marked entry. Libprefetch adjusts its idea of the application's current position based on the new list. The number of accepted entries is returned; once libprefetch's access-list buffer fills up, this will be less than the number passed in. When the application has transferred its entire access list to libprefetch, or libprefetch has indicated that its buffer is full, the application signals that it is done updating the list.

Libprefetch assumes that the sequence of file reads within registered file regions is complete—all read requests to registered regions should correspond to access-list entries. Accesses to non-registered regions are ignored; applications can manage these regions with other mechanisms. If the application accesses a file offset within a registered region but not in the access list, libprefetch assumes the application has changed the access plan and issues a callback to update the access list. However, the application can tell libprefetch to ignore a series of accesses. This is useful to avoid callback recursion: sometimes a callback must itself read prefetchable data while calculating the upcoming access list.

4.3 Callback Example

Figure 7 presents pseudocode for a sample callback, demonstrating how libprefetch is used. This callback is similar to the one used in our GIMP benchmark (Section 5.3). GIMP divides images into square regions of pixels called *tiles*; during image transformations, it iterates through the tiles in both row- and column-major order, both of which could cause nonsequential access. The pseudocode prefetches an image's tile data in the current access order (img->accessOrder).

The callback first uses its arguments to determine the application's position in its access pattern (lines 4–5). Next, it traverses tile information structures to determine future accesses (line 6). For each future access, the callback records the file descriptor, offset, and whether the access is considered marked (lines 7–9). Access entries accumulate in an array and are passed to libprefetch in batches (line 10). If libprefetch's access-list buffer fills up, the callback returns (lines 12, 15–16). The callback's first call to **request_prefetching** clears the old access list and sets it to the new value (line 3); subsequent calls append to the access list under construction (line 13). Finally, the callback informs libprefetch about any remaining access entries and signals completion (line 18).

```
void callback(state, markFd, markOffset, reqFd, reqOffset) {
    struct access_entry accesses[BATCH_SIZE];
    int accepted, full, n = 0;
    int mode = PF_SET;
    tileInfo_t *tile = getTileInfo(reqFd, reqOffset);
    imageInfo_t *img = tile->imageInfo;

    for (; !lastTile(tile); tile = nextTile(tile, img->accessOrder) ) {
        accesses[n].page_offset = tile->swap_offset;
        accesses[n].fd = img->swap_file;
        accesses[n++].marked = 0;
        if (n == BATCH_SIZE) {
            accepted = request_prefetching(state->client,
                accesses, n, mode);
            full = (accepted < n);
            mode = PF_APPEND;
            n = 0;
            if (full)
                break;
    }   }
    request_prefetching(state->client, accesses, n,
        mode | PF_DONE);
}
```

Figure 7: Pseudocode for a libprefetch callback function.

4.4 Interface Discussion

We evaluated several alternatives before arriving at the libprefetch interface. Previous designs' deficiencies may illuminate libprefetch's virtues.

The initial version of libprefetch stored several flags for each access-list entry, including a flag that indicated the page would be used only once (it should be evicted immediately after use). In several cases, after some unproductive debugging, we found that we had incorrectly set the flag. By tracking progress through the access list, the next iteration of libprefetch made it possible to automatically detect pages that aren't useful in the short-term future, so we removed this ability to accidentally induce poor performance.

An earlier attempt to enhance prefetching tried to expand the methodology used by readahead, namely inferring future accesses from current accesses. Readahead infers future accesses by assuming that several sequential accesses will be followed by further sequential accesses. Our extension of this concept, **fdepend**, allowed the application to make explicit the temporal relationships between different regions of files. With **fdepend**, an application might inform the kernel that after accessing data in range *A*, it would access data in range *B*, but no longer access data in range *C*. This mechanism introduced problems that we later solved in libprefetch. For example, because it wasn't clear which relationships would be useful in advance of a particular instant of execution, applications would specify all relationships up front. This caused large startup delays as an application enumerated all the relationships; in addition, storing all the relationships required substantial memory. Libprefetch's access list is both simpler for applications to generate and easier to specify incrementally.

4.5 Implementation

Libprefetch is mostly implemented as a user-level library. This choice both demonstrates the benefits possible with minimal kernel changes and avoids end-runs around our operating system's existing caching and prefetching policies. An application using libprefetch might issue different file-system-related system calls than the unmodified application, but as far as the operating system is concerned, it is doing nothing out of the ordinary. The kernel need not change its policy for managing different applications' conflicting needs. (Nevertheless, a kernel implementation could integrate further with existing code, would have better access to buffer cache state and file system layout, and might offer speed advantages by reducing system call overhead.) Infill, however, is implemented as a kernel modification. Infill is not strictly a prefetching optimization, but a faster way to read specific patterns of blocks.

Each time libprefetch intercepts a **read** from the application, it first checks whether the read corresponds to a registered region. If so, it uses a new system call, **fincore**, to see whether the requested page(s) are already in the buffer cache. If there is a miss in the buffer cache, a page has been prematurely evicted or the application has strayed from its access list. In either case, libprefetch issues a new round of prefetch requests, possibly calling into the application first to update the access list. The **fincore** system call was inspired by **mincore**; it takes a file descriptor, an offset and length, and the address of a bit vector as input, and fills in the bit vector with the state of the requested pages of the file (in memory or not).

When libprefetch decides that it should prefetch more data, it first consults the contention controller (described below) to determine the size of its reorder buffer for this round of prefetching. Then it walks the access list until it has seen the appropriate number of unique pages. Once the set of pages to be prefetched is determined, libprefetch evicts any pages from the previous round of prefetching that are not in the current prefetch set and asks the kernel to prefetch, in file offset order, the new set of pages. This process is where libprefetch differs most from previous prefetching systems. Instead of overlapping I/O and CPU time, libprefetch blocks the application while it fetches many disk blocks. This makes sense, particularly for nonsequential access patterns, because sorted and batched requests are usually faster than in-order requests. Prefetching from a separate thread would allow the main thread to continue once its next request was in memory, but as soon as the main thread made a request located near the end of the sorted reorder buffer, it would block for the rest of the prefetching phase anyway.

Libprefetch makes prefetch requests using **posix_-fadvise**, a rarely-used system call that does for files what **madvise** does for memory. It takes a file descriptor, an offset and length, and "advice" about that region of the file. Libprefetch uses two pieces of advice: POSIX_FADV_WILLNEED informs the kernel that the given range of the file should be brought into the buffer cache, and POSIX_FADV_DONTNEED informs it that the given range of the file is no longer needed and can be dropped from the buffer cache.

We found some weaknesses in the Linux implementation of **posix_fadvise**. The WILLNEED advice has no effect on file data that is already in memory; for example, if a given page is next on the eviction list before WILLNEED advice, it will still be next on the eviction list after that advice. We therefore changed **posix_fadvise** to move already-in-memory pages to the same place in the LRU list they would have been inserted had they just come from disk. The implementation of DONTNEED also has pitfalls. Linux suggests applications flush changes before issuing DONTNEED advice because dirty pages are not guaranteed to be evicted. However, it tries to assist with this requirement by starting an asynchronous write-back of dirty data in the file upon receiving DONTNEED advice. Unfortunately, it starts this write-back even if the to-be-evicted data is not dirty. Because of this unexpected behavior, we found it faster to DONTNEED pages in large batches instead of incrementally.

Our modifications supply **posix_fadvise** with additional functionality: libprefetch uses the system call to intentionally reorder the buffer cache's eviction list. Libprefetch already uses DONTNEED advice to discard pages it no longer needs, but ordering the eviction list further improves performance in the face of memory pressure. After prefetching a set of pages in disk order, libprefetch again advises the kernel that it WILLNEED that data, but in reverse access order. If memory pressure causes some pages to be evicted, the evicted pages will now be those needed furthest in the future. This enables the application to make as much progress as possible before libprefetch must re-prefetch evicted pages.[3]

Together, **fincore** and our modified **posix_fadvise** give libprefetch enough access to buffer cache state to work effectively. **fincore** lets libprefetch query the state of particular pages, and **posix_fadvise** lets it bring in pages from disk, evict them from the buffer cache, and reorder the LRU list. Since these operations are all done from user space, existing kernel mechanisms can account for resource usage and provide fairness among processes. An efficient system call that translated file offsets to the corresponding on-disk block numbers would improve libprefetch's support for nonsequential file layouts.

Libprefetch is approximately 2,400 lines of commented code, including several disabled features that showed no benefit. The kernel changes for **posix_fadvise** and **fincore** are 130 lines long.

4.6 Concurrent Execution

As described so far, libprefetch monopolizes both disk bandwidth and the buffer cache. Of course, this behavior could seriously degrade other applications' performance. In this section we discuss modifications to libprefetch that improve fairness and application performance even with multiple uncoordinated applications running concurrently.

Disk contention is the easier problem to solve: Linux's default fairness mechanisms work effectively as is. From the point of view of the operating system, disk contention caused by libprefetch is indistinguishable from contention caused by any other application. Further disk access coordination could yield better performance, but might raise other issues, such as fairness, denial of service, and security concerns.

The buffer cache, however, requires a different approach. As discussed, libprefetch should use the maximum buffer cache space available. An early implementation simply queried the operating system for the amount of buffer cache space and used half of it (half showed the best performance in experiments). However, this did not account for other applications using the buffer cache, and in benchmarks with more than one application libprefetch's attempt to dominate the buffer cache severely degraded performance. The current libprefetch explicitly addresses buffer-cache contention, but manages to do so without explicit coordination among processes. The key insight is that buffer cache management can be reformulated as a congestion-control problem. Libprefetch uses an additive-increase, multiplicative-decrease (AIMD) strategy, also used by TCP for network congestion control, to adapt to changes in available memory.

Libprefetch infers a contention signal when it finds that some of the pages it prefetched have been prematurely evicted. This should happen only under memory pressure, so libprefetch lowers its reorder buffer size with a multiplicative decrease. Conversely, when libprefetch consumes all of the pages it prefetched without any of them being prematurely evicted, it increases its reorder buffer size by an additive constant.

Because libprefetch knows the maximum size of the buffer cache (from the **/proc** file system), it starts out using most of available cache space, instead of using a slow-start phase. We always limit libprefetch's reorder buffer to 90% (experimentally determined) of the RAM available for the buffer cache. Furthermore, since the contention controller can only adjust the reorder buffer size during a round of prefetching, it is somewhat aggressive in its upward adjustment, adding 10% of the maximum buffer cache space to the reorder buffer size; it halves the reorder buffer size to decrease. A quick evaluation of alternative values shows that libprefetch is not particularly sensitive to AIMD constants. The result performs well: concurrently-running libprefetch-enabled applications transparently coordinate to achieve good performance, and libprefetch applications do not significantly degrade the performance of other applications.

4.7 Disk Request Infill

To implement infill, we modified the general Linux I/O scheduler framework and its pluggable CFQ scheduler (about 800 lines of changes). Linux's I/O schedulers already have the ability to merge adjacent requests, but they cannot merge non-adjacent ones. Upon receiving a new request, Linux looks for a queued request to merge with the incoming one. If no request is found, our modified scheduler then looks for the queued request nearest the new one. If the nearest request is within the maximum infill distance, we create a dummy request and merge it with the queued request. The incoming request is now adjacent to the expanded request and the two are merged.

While the I/O scheduler has infill requests in its queue, a new request may arrive that overlaps an infill request. The scheduling framework did not handle overlapped requests, so we explicitly link new requests to any queued infill requests that they overlap. When the infill request completes, the new request is serviced from the infill request's data.

5 Evaluation

In this section we evaluate libprefetch's impact on three different kinds of workloads: sequential, "strided," and nonsequential. Linux's readahead mechanism already does a good job of optimizing sequential workloads; for these we expect at most a modest improvement with libprefetch. A *strided* workload consists of groups of sequential accesses separated by large seeks. Linux does not specifically try to detect strided access patterns and does little to improve their performance (unless the run of sequential accesses is substantial). Readahead is of no help to nonsequential access patterns, and we expect the biggest improvement with this workload. Our strided and nonsequential access patterns come from benchmarks of real applications, namely the GNU Image Manipulation Program (GIMP) and the SQLite database. This shows that libprefetch significantly improves real application performance.

When evaluating these workloads, we vary the amount of data accessed relative to the amount of RAM, showing how relative reorder buffer size affects prefetching improvements. In addition, we examine the impact of infill and the performance of multiple uncoordinated applications running concurrently.

5.1 Methodology

All the benchmarks in this section were run on a Dell Precision 380 with a 3.2GHz Pentium 4 CPU with 2MB of L2 cache (hyperthreading disabled), a Silicon Image 3132-2 SATA Controller, and 512MB of RAM. Log output was written to another machine via sshfs. Tests use a modified Linux 2.6.20 kernel on the Ubuntu v8.04 distribution. The small size of main memory was chosen so that our tests of stock software would complete in a reasonable amount of time (a single stock 4GB SQLite test takes almost ten hours). As mentioned in Section 3.5, we believe libprefetch's speedup relative to unoptimized accesses is constant for a given ratio of data set size to prefetching memory, at least for uniform random accesses. Unless otherwise noted, these tests used Disk 1, a 500GB 7200 RPM SATA2 disk with a 32MB buffer and 8.5ms average seek time (Seagate ST3500320AS, firmware SD1A). Disk 2 is Disk 1 with older firmware, version SD15.

Some of the benchmarks in Section 3 used additional disks. Disk 3 is a 500GB 7200 RPM SATA2 disk with a 16MB buffer and 8.9ms average seek time (Western Digital WD5000AAKS) in a HP Pavilion Elite D5000T with a 2.66GHz Core 2 Quad Q9450 and 8GB of RAM, and Disk 4 is a 320GB 7200 RPM SATA2 disk with a 16MB buffer and 8.5ms average seek time (Seagate ST3320620AS) in a Dell Optiplex GX280 with a 2.8GHz Pentium 4 CPU, 1MB of L2 cache (hyperthreading disabled), a Silicon Image 3132-2 SATA Controller, and 512MB of RAM.

5.2 Sequential Access

Our sequential benchmark program reads a file from beginning to end. It is similar to **cat *file*** > **/dev/null**, but is libprefetch-enabled and can change various Linux readahead options. We observed, and confirmed with **dd**, great variance in sequential read performance on different files, from 20MB/s to 110MB/s. These differences are due to file fragmentation. For example, the fastest file has a significantly longer average consecutive block run

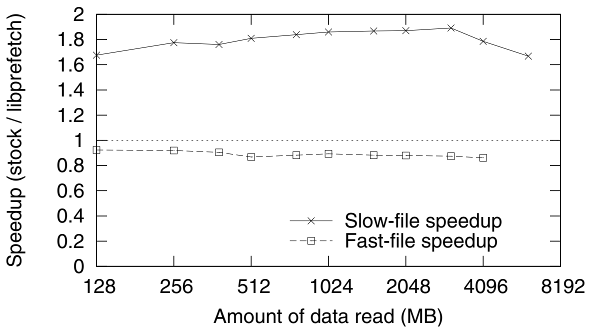


Figure 8: Speedup with libprefetch when reading a file sequentially. Due to file layout issues, some files can be retrieved more quickly; this benchmark shows both a fast and slow file. The slowdown on the fast file, 6 seconds for the 4GB test, is mostly due to libprefetch's CPU overhead. The slow file runtimes range from 6 s to 365 s for stock and 6 s to 220 s for libprefetch.

than the slowest file (3.8MB vs. 14KB).

For the slowest file, Figure 8 shows that libprefetch achieves improvements similar to previous prefetching work. We believe that Linux readahead is slower than libprefetch in this case because libprefetch sends many more requests to the disk scheduler at once, giving it more opportunity to reorder and batch disk requests. With the fast file, libprefetch is slightly slower than readahead. Examining the system and user time for the tests shows that the majority of the difference can be attributed to the additional CPU overhead that libprefetch incurs, which we have not yet tried to optimize.

5.3 Strided Access

Our strided benchmarks use the GNU Image Manipulation Program (GIMP) to blur large images, a workload similar to common tasks in high-resolution print or film work. GIMP divides the image into square tiles and processes them in passes, either by row or by column. When GIMP's memory requirement for tiles grows beyond its internal cache size, it overflows them to a swap file. The swap file access patterns manifest themselves as strided disk accesses when a row pass reads the output of a column pass or vice versa. To blur an image, GIMP makes three strided passes over the swap file. While it is feasible to correctly detect and readahead these kinds of access patterns, Linux does not attempt to do so.

We changed the GIMP tile cache to allow processing functions to declare their access patterns: they specify the order (row or column) in which they will process a set of images. Multiple image passes are also expressible. We exposed this interface to GIMP plugins and modified the blur plugin to use this infrastructure (the modification was simple). The core GIMP functions use a common abstraction to make image passes, which we also modified to use the access pattern infrastructure. We changed a total of 679 lines: 285 for the plugin architecture, 40 to specify patterns in the blur plugin, 11 to alter the core image pass abstraction, and 343 to implement the libprefetch callback.

We benchmarked the time to blur a square RGBA image of the given size. Blur uses two copies of the image for most of the operation and three to finish, so memory requirements are higher than just the raw image size. We set the GIMP's internal cache to 100MB to prevent significant amounts of double buffering in the GIMP and the operating system's buffer cache; we do not use less than 100MB so that GIMP's internal cache can contain up to three working image rows or columns for our largest test image. GIMP mallocs space for its file cache, so libprefetch does not attempt to use any of that 100MB of memory per GIMP instance. The GIMP benchmark is read/write, whereas our other benchmarks are read-only.

The results are shown in Figure 9. When the image size is small, all the data is held in the GIMP's internal cache and the operating system's buffer cache, so there is no disk access to optimize; stock and libprefetch runtimes are equal. As the image size increases from 192MB to 1GB, disk access increases and libprefetch achieves greater speedups. Libprefetch retrieves data from multiple rows (columns) before striding to the next part of those rows (columns). This amortizes the cost of the strided access pattern across the retrieval of multiple rows (or columns), achieving a speedup of up to 5x. As the image size increases, however, the number of rows or columns that can be retrieved in one pass decreases. The results for images greater than 1GB in size show this gradual decrease in speedup.

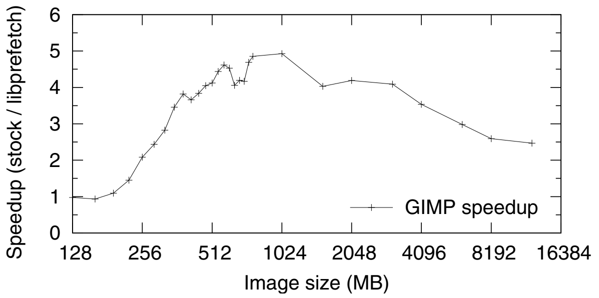


Figure 9: Speedup with libprefetch for GIMP to blur various sized images. When GIMP uses the disk, libprefetch reduces runtime by a factor of 2 to 5. Stock runtimes range from 28 seconds for the 128MB workload to almost 7.5 hours for the 12GB workload; the intermediate size of 1GB takes 38 minutes. The libprefetch runtime also starts at 28 seconds, but only climbs to 3 hours; the 1GB size takes 5 minutes.

5.4 Nonsequential Access

Our nonsequential benchmark issues a query to a SQLite database. The dataset in the database is TPC-C like [6] with the addition of a secondary index by zip code on the **customer** table. We used datasets with 7 to 218 warehouses, yielding sizes between 132MB and 4110MB for the combination of the customer table and zip code index. Additionally, we configured SQLite to use 4KB pages (instead of the default 1KB pages) to match the storage unit and reduce false sharing.

The benchmark performs the query **SELECT * FROM customer ORDER BY c_zip**. (Runtime on this query was within a few percent on stock SQLite and stock MySQL.) For this query, each resulting row will be in a random file location relative to the previous row, inducing a large number of seeks. Consequently, we expect the query to have poor performance. When the dataset fits entirely in memory, each disk page only needs to be read once, after which the rest of the workload will be serviced from the buffer cache. However, if the dataset is larger than memory, pages will be read from disk multiple times (each page holds multiple rows).

The SQLite callback for libprefetch examines the off-

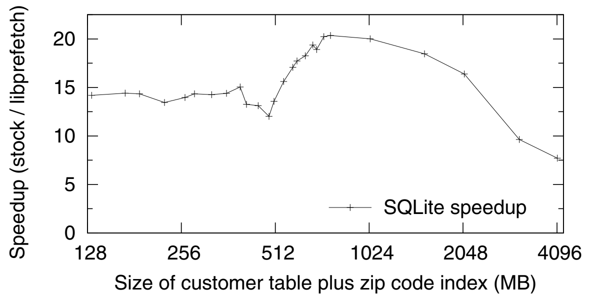


Figure 10: Speedup with libprefetch for SQLite when scanning a table by a secondary index. The initial 14x approximate speedup peaks at 20.3x, then falls to 7.7x. Stock runtime starts at about 3 minutes, climbs to 100 minutes by the 1GB test, and runs for nearly 10 hours for the 4GB test. Libprefetch's runtime starts at 12 seconds and is only 77 minutes for the 4GB test; the 1GB test takes 7.5 minutes.

set for the current position and determines if it belongs to an index. If so, the callback iterates the index and tells libprefetch about the table data that the index points to. From then on, the callback marks the index pages and uses them to track its progress. This modification adds less than 500 lines of code.

Figure 10 shows the speedups for the nonsequential SQLite benchmark with 132MB to 4GB of data. The initial improvement of roughly 14x is because libprefetch is able to load the entire dataset in sequential order, whereas SQLite loads the data on demand (in random order) as it traverses the zip code index. As the dataset approaches the size of memory, the speedup decreases because libprefetch starts to require multiple passes over the disk. Then, between roughly 512MB and 768MB, we see a sharp increase in speedup: stock SQLite requires progressively more time due to a sharp decline in the buffer cache hit ratio as the dataset size exceeds the size of memory. As the dataset grows even larger, the density of libprefetch's passes over the disk decreases, causing higher average seek distance and decreasing benefits from libprefetch.

Libprefetch processed the 4GB data set in just under 77 minutes. This is roughly 400 times slower than the 128MB data set, which took 11.6 s; processing time increased by about 13x more than data set size. Some expensive seeks are simply unavoidable. However, the libprefetch time is still 7.7x faster than stock SQLite.

5.5 Infill

Infill has no significant effect on the sequential or strided benchmarks because they have few infill opportunities. The large seeks in the strided access pattern are too large for infill to be a win. Similarly, stock SQLite with infill shows no significant speedup; the gaps between most requests are too large for infill to apply.

The effect of infill on SQLite when using libprefetch is shown in Figure 11. Disk 2 shows a substantial improvement for many of the tests, up to 2.3x beyond the speedups that libprefetch achieves. Libprefetch's reordering shrinks the average gap between requests to the point where infill can improve performance. But as the dataset size increases, the density of requests in a given libprefetch pass across the disk decreases; as a result, infill's applicability also decreases.

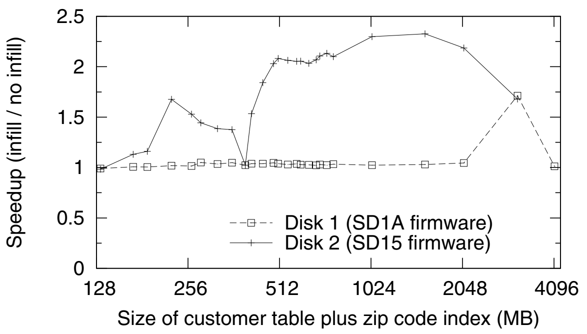


Figure 11: Speedup due to infill for Disk 1 and Disk 2. All tests used libprefetch. A firmware upgrade mostly alleviated the need for infill, though a modest effect is still observed.

As noted earlier, Disk 1 and Disk 2 are the same disk: Disk 1 has newer firmware. While there is a noticeable difference in the infill speedup on these two disks, the difference in runtime when both libprefetch and infill are used is less than 10%. It appears that the updated firmware takes advantage of the hardware effect that leads to infill being useful. The 3GB test is somehow an exception, achieving a 1.7x speedup from infill on Disk 1. Except for that point, the maximum speedup from infill for Disk 1 is 1.049x. That is on par with the maximum infill speedup we saw on Disk 3, 1.084x.

Infill will never be helpful for some access patterns because there simply isn't any opportunity to apply it. For other access patterns, when infill is used with libprefetch, it can either dramatically increase performance or provide a modest improvement, depending on the disk. None of our experiments showed a substantial negative impact from infill.

5.6 Concurrent Applications

We evaluated libprefetch's effect on concurrent workloads first by running multiple concurrent instances of our benchmarks. Each instance used its own data file, so prefetching in one instance didn't help any other. Figure 12 shows the runtime for 1 to 3 concurrent executions of both the SQLite and GIMP benchmarks (512MB datasets). Both with and without libprefetch, the runtime of the GIMP benchmark scales with the number of instances. The libprefetch versions run 3x to 4x faster than the stock versions. Stock SQLite scales more slowly than the number of instances; two SQLite instances take over 7x as long as a single instance, and three take more than 13x as long as one. Libprefetch improves SQLite's scal-

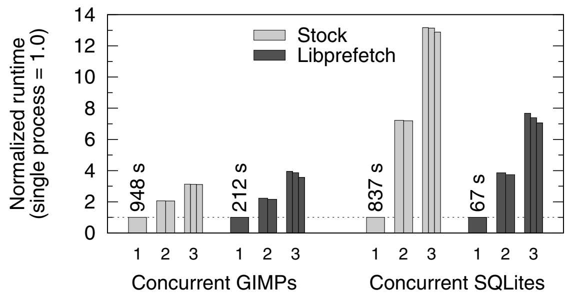


Figure 12: Stock vs. libprefetch application performance for one, two, and three application instances running concurrently. To highlight scaling behavior, times within each application group are normalized to the performance of a single application instance.

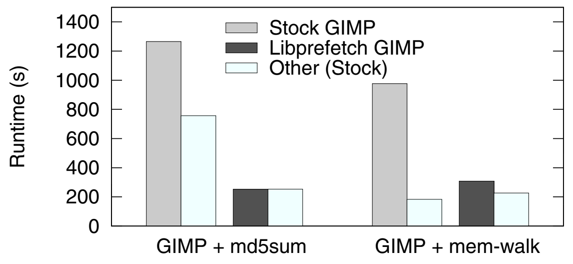


Figure 13: Runtime of GIMP, with and without libprefetch, measured concurrently with CPU- and memory-intensive microbenchmarks.

ing behavior; two SQLites take nearly 4x as long as one, and three take almost 8x as long as one. The libprefetch speedup over stock SQLite is 23.5x for two and 21.5x for three concurrent instances.

While libprefetch SQLite scales better than stock, libprefetch GIMP does not. We believe this is due to the amount of memory available for reorder buffers. Whereas each SQLite process occupies about 20MB of memory, each GIMP process occupies about 150MB (predominantly for its tile cache). On a machine with 500MB of memory available after bootup and three test processes, the memory available for reorder buffers is less than 50MB for GIMP versus 440MB for SQLite.

We also confirmed that libprefetch's AIMD contention controller has the intended effect. With the contention controller disabled, each libprefetch process tried to use the entire buffer cache, causing many pages to be evicted before use. The tests ran for several times the stock runtime before we we gave up and killed them. The AIMD mechanism is effective and necessary with our approach to contention management.

We tested resource contention more directly by running GIMP and SQLite concurrently with two resource-heavy benchmarks, md5sum and mem-walk. Md5sum calculates the MD5 checksum of a 2.13GB file; mem-walk allocates 100MB of memory and then reads each page in turn, cycling through the pages for a specified number of iterations. Figure 13 shows the runtime for these two microbenchmarks run concurrently with GIMP, both with and without libprefetch. When running md5sum and GIMP concurrently, md5sum is faster with the libprefetch-enabled version of the GIMP. This is because the libprefetch-enabled GIMP has lower disk utilization, yielding more time for md5sum to use the disk.

An opposite effect comes into play when running the GIMP concurrently with the mem-walk benchmark. Since the same amount of CPU time is spent over a shorter total time, GIMP with libprefetch has a higher CPU utilization. Mem-walk takes about 25% longer with the libprefetch-enabled GIMP because it is scheduled less frequently. This slowdown is not specific to libprefetch; any CPU-intensive application would have a similar effect. The libprefetch speedup that GIMP gets with mem-walk is not as high as with md5sum, partly due to higher CPU contention and partly due to the smaller amount of memory available to libprefetch. Results for md5sum and mem-walk run concurrently with SQLite are similar.

6 Conclusion

An analysis of the performance characteristics of modern disks led us to a new approach to prefetching. Our prefetching algorithm minimizes the number of expensive seeks and leads to a substantial performance boost for nonsequential workloads. Libprefetch, a relatively simple library that implements this technique, can speed up real-world instances of nonsequential disk access, including image processing and database table scans, by as much as 4.9x and 20x, respectively, for workloads that do not fit in main memory. Furthermore, a simple contention controller enables this new prefetching algorithm to peacefully coexist with multiple instances of itself as well as other applications.

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback, and our shepherd, Geoff Kuenning, who was very generous with his time. This work was supported by the National Science Foundation under grants NSF-0430425, NSF-0427202, and NSF-0546892. Eddie Kohler is also supported by a Microsoft Research New Faculty Fellowship and a Sloan Research Fellowship.

Notes

[1]For instance, the Apple Hard Disk 20SC, introduced in 1985, had an average access time of 65 to 85 msec and a maximum transfer speed of 1.25 MB/s [11]. The specifications for our disk (Seagate Barracuda 7200.11) quote an average access time of 4.16 msec and sustained transfer speed up to 105 MB/s [24].

[2]This formula was derived using uniform random real numbers, ignoring quantization effects, and is most precise when $5 \leq K \ll N$.

[3]Because prefetching and readahead are speculative, prefetched pages are inserted into the LRU list at a lower priority (at the head

of the inactive list) than pages that were explicitly **read**. This can lead to discrepancies between the LRU list order and the access list order.

References

[1] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, May 2001.

[2] Pei Cao, Edward Felten, Anna Karlin, and Kai Li. Implementation and performance of integrated application-controlled caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, Louisiana, February 1999.

[4] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Department of Computer Science Tech Report PCS-TR94-243, Dartmouth College, November 1994.

[5] Microsoft Corp. *SuperFetch*, 2006. `http://www.microsoft.com/windows/windows-vista/features/superfetch.aspx`.

[6] Transaction Processing Performance Council. *TPC-C Online Transaction Processing Benchmark*, April 2009. `http://www.tpc.org/tpcc/`.

[7] Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proc. 2003 USENIX Annual Technical Conference*, pages 325–338, San Antonio, Texas, June 2003.

[8] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proc. USENIX Summer 1994 Technical Conference*, pages 197–207, Boston, MA, June 1994.

[9] IEEE. *POSIX 1003.1-2001*, 2001.

[10] IEEE. *POSIX 1003.1b*, 1993.

[11] Apple Inc. Apple Hard Disk 10SC: Specifications (Discontinued), November 2008. `http://docs.info.apple.com/article.html?artnum=1931`.

[12] Apple Inc. HFS plus volume format. Section: Hot files. Technical Note TN1150, March 2004.

[13] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian N. 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 *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 19–34, Seattle, Washington, October 1996.

[14] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 14–19, Rio Rico, AZ, March 1999.

[15] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proc. 1997 USENIX Annual Technical Conference*, pages 275–288, Anaheim, California, January 1997.

[16] Mark Palmer and Stanley B. Zdonik. FIDO: A cache that learns to fetch. In *Proc. 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 255–264, Barcelona, Catalonia, Spain, September 1991.

[17] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems (PDIS '94)*, pages 7–16, Austin, TX, 1994.

[18] R. Hugo Patterson, Garth A. Gibson, Eka Gintin, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[19] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[20] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 225–238, San Francisco, CA, December 2005.

[21] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[22] Michael Stonebraker, John Woodfill, Jeff Ranstrom, Marguerite Murphy, Marc Meyer, and Eric Allman. Performance enhancements to a relational database system. *ACM Transactions on Database Systems*, 8(2):167–185, June 1983.

[23] Carl Tait and Dan Duchamp. Detection and exploitation of file working sets. In *Proc. 11th International Conference on Distributed Computing Systems*, pages 2–9, Arlington, TX, May 1991.

[24] Seagate Technology. ST3500320AS - Barracuda 7200.11 SATA 3Gb/s 500-GB Hard Drive, April 2009. `http://www.seagate.com/ww/v/index.jsp?vgnextoid=c89ef141e7f43110VgnVCM100000f5ee0a0aRCRD`.

[25] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. *ACM SIGARCH Computer Architecture News*, 22(4):23–28, September 1994.

[26] Kishor S. Trivedi. An analysis of prepaging. *Computing*, 22(3): 191–210, September 1979.

[27] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.