5-2000

Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives (CMU-CS-00-130)

C. Lumb
Carnegie Mellon University

J. Schindler
Carnegie Mellon University

G. R. Ganger
Carnegie Mellon University

D. F. Nagle
Carnegie Mellon University

E. Riedel
Carnegie Mellon University

Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI), October 2000.

Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives

Christopher R. Lumb, Jiri Schindler,
Gregory R. Ganger, David F. Nagle
Carnegie Mellon University
Erik Riedel
Hewlett-Packard Labs

Abstract

Freeblock scheduling is a new approach to utilizing more of a disk's potential media bandwidth. By filling rotational latency periods with useful media transfers, 20–50% of a never-idle disk's bandwidth can often be provided to background applications with no effect on foreground response times. This paper describes freeblock scheduling and demonstrates its value with simulation studies of two concrete applications: segment cleaning and data mining. Free segment cleaning often allows an LFS file system to maintain its ideal write performance when cleaning overheads would otherwise reduce performance by up to a factor of three. Free data mining can achieve over 47 full disk scans per day on an active transaction processing system, with no effect on its disk performance.

1 Introduction

Disk drives increasingly limit performance in many computer systems, creating complexity and restricting functionality. However, in recent years, the rate of improvement in media bandwidth (40+% per year) has stayed close to that of computer system attributes that are driven by Moore's Law. It is only the mechanical positioning aspects (i.e., seek times and rotation speeds) that fail to keep pace. If 100% utilization of the potential media bandwidth could be realized, disk performance would scale roughly in proportion to the rest of the system over time. Unfortunately, utilizations of 2–15% are more commonly observed in practice.

This paper describes and analyzes a new approach, called *freeblock scheduling*, to increasing media bandwidth utilization. By interleaving low priority disk activity with the normal workload (here referred to as background and foreground, respectively), one can replace many foreground rotational latency delays with useful background media transfers. With appropriate freeblock scheduling, background tasks can receive 20–50% of a disk's potential media bandwidth without any increase in foreground request service times. Thus, this background disk activity is completed for free during the mechanical positioning for foreground requests.

There are many disk-intensive background tasks that are designed to occur during otherwise idle time. Examples include disk reorganization, file system cleaning, backup, prefetching, write-back, integrity checking, RAID scrubbing, virus detection, tamper detection, report generation, and index reorganization. When idle time does not present itself, these tasks either compete with foreground tasks or are simply not completed. Further, when they do compete with other tasks, these background tasks do not take full advantage of their relatively loose time constraints and paucity of sequencing requirements. As a result, these "idle time" tasks often cause performance or functionality problems in busy systems. With freeblock scheduling, background tasks can operate continuously and efficiently, even when they do not have the system to themselves.

This paper quantifies the effects of disk, workload, and disk scheduling algorithms on potential free bandwidth. Algorithms are developed for increasing the available free bandwidth and for efficient freeblock scheduling. For example, with less than a 6% increase in average foreground access time, a Shortest-Positioning-Time-First scheduling algorithm that favors reduction of seek time over reduction of rotational latency can provide an additional 66% of free bandwidth. Experiments also show that freeblock scheduling decisions can be made efficiently enough to be effective in highly loaded systems.

This paper uses simulation to explore freeblock scheduling, demonstrating its value with concrete

examples of its use for storage system management and disk-intensive applications. The first example shows that cleaning in a log-structured file system can be done for free even when there is no truly idle time, resulting in up to a 300% speedup. The second example explores the use of free bandwidth for data mining on an active on-line transaction processing (OLTP) system, showing that over 47 full scans per day of a 9GB disk can be made with no impact on OLTP performance.

In a recent paper [45], we proposed a scheme for performing data mining "for free" on a busy OLTP system. The scheme combines Active Disks [46] with use of idle time and aggressive interleaving of data mining requests with OLTP requests. This paper generalizes and extends the latter, developing an understanding of free bandwidth availability and exploring its use.

The remainder of this paper is organized as follows. Section 2 describes free bandwidth and discusses its use in systems. Section 3 quantifies the availability of potential free bandwidth and how it varies with disk characteristics, foreground workloads, and foreground disk scheduling algorithms. Section 4 describes our freeblock scheduling algorithm. Section 5 evaluates the use of free bandwidth for cleaning of LFS log segments. Section 6 evaluates the use of free bandwidth for data mining of active OLTP systems. Section 7 discusses related work. Section 8 summarizes the paper's contributions.

2 Free Bandwidth

At a high-level, the time required for a disk media access, T_{access}, can be computed as

$$T_{access} = T_{seek} + T_{rotate} + T_{transfer}$$

Of T_{access}, only the $T_{transfer}$ component represents useful utilization of the disk head. Unfortunately, the other two components generally dominate. Many data placement and scheduling algorithms have been devised to increase disk head utilization by increasing transfer sizes and reducing positioning overheads. Freeblock scheduling complements these techniques by transferring additional data during the T_{rotate} component of T_{access}.

Fundamentally, the only time the disk head cannot be transferring data sectors to or from the media is during a seek. In fact, in most modern disk drives, the firmware will transfer a large request's data to or from the media "out of order" to minimize wasted time; this feature is sometimes referred to as zero-latency or immediate access. While seeks are unavoidable costs associated with accessing desired data locations, rotational latency is an artifact of not doing something more useful with the disk head. Since disk platters rotate constantly, a given sector will rotate past the disk head at a given time, independent of what the disk head is doing up until that time. So, there is an opportunity to do something more useful than just waiting for desired sectors to arrive at the disk head.

Freeblock scheduling consists of predicting how much rotational latency will occur before the next foreground media transfer, squeezing some additional media transfers into that time, and still getting to the destination track in time for the foreground transfer. The additional media transfers may be on the current or destination tracks, on another track near the two, or anywhere between them, as illustrated in Figure 1. In the two latter cases, additional seek overheads are incurred, reducing the actual time available for the additional media transfers, but not completely eliminating it.

Accurately predicting future rotational latencies requires detailed knowledge of many disk performance attributes, including layout algorithms and time-dependent mechanical positioning overheads. These predictions can utilize the same basic algorithms and information that most modern disks employ for their internal scheduling decisions, which are based on overall positioning overheads (seek time plus rotational latency) [51, 30]. However, this may require that freeblock scheduling decisions be made by disk firmware. Fortunately, the increasing processing capabilities of disk drives [1, 22, 32, 46] make advanced on-drive storage management feasible [22, 57].

2.1 Using Free Bandwidth

Potential free bandwidth exists in the time gaps that would otherwise be rotational latency delays for foreground requests. Therefore, freeblock scheduling must opportunistically match these potential free bandwidth sources to real bandwidth needs that can be met within the given time gaps. The tasks that will utilize the largest fraction of potential free bandwidth are those that provide the freeblock scheduler with the most flexibility. Tasks that best fit the freeblock scheduling model have low priority, large sets of desired blocks, no particular order of access, and small working memory footprints.

Low priority. Free bandwidth is inherently in the background, and freeblock requests will only be serviced when opportunities arise. Therefore, response times may be extremely long for such requests, mak-

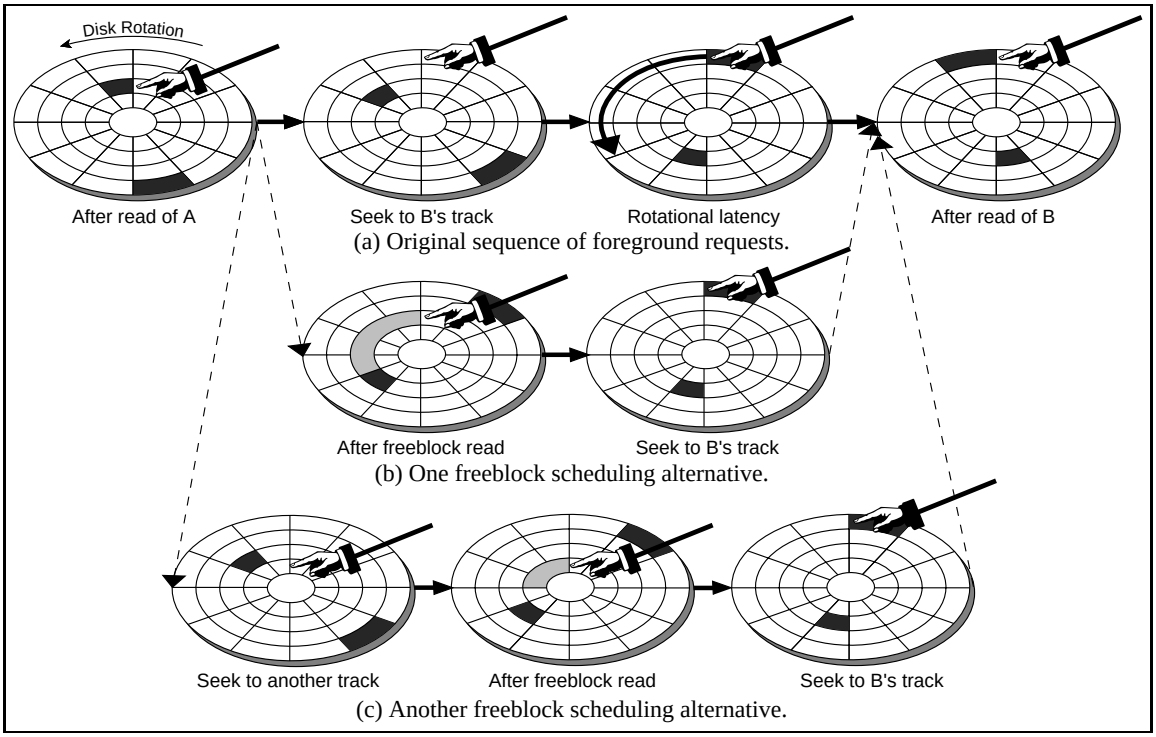


Figure 1: **Illustration of two freeblock scheduling possibilities.** Three sequences of steps are shown, each starting after completing the foreground request to block A and finishing after completing the foreground request to block B. Each step shows the position of the disk platter, the read/write head (shown by the pointer), and the two foreground requests (in black) after a partial rotation. The top row, labelled (a), shows the default sequence of disk head actions for servicing request B, which includes 4 sectors worth of potential free bandwidth (a.k.a. rotational latency). The second row, labelled (b), shows free reading of 4 blocks on A's track using 100% of the potential free bandwidth. The third row, labelled (c), shows free reading of 3 blocks on another track, yielding 75% of the potential free bandwidth.

ing them most appropriate for background activities. Further, freeblock scheduling is not appropriate for a set of equally important requests; splitting such a set between a foreground queue and a freeblock queue reduces the options of both schedulers. All such requests should be considered by a single scheduler.

Large sets of desired blocks. Since freeblock schedulers work with restricted free bandwidth opportunities, their effectiveness tends to increase when they have more options. That is, the larger the set of disk locations that are desired, the higher the probability that a free bandwidth opportunity can be matched to a need. Therefore, tasks that involve larger fractions of the disk's capacity generally utilize larger fractions of the potential free bandwidth.

No particular order of access. Ordering requirements restrict the set of requests that can be considered by the scheduler at any point in time. Since the effectiveness of freeblock scheduling is directly related to the number of outstanding requests, workloads with little or no ordering requirements tend to utilize more of the potential free bandwidth.

Small working memory footprints. Significant need to buffer multiple blocks before processing them creates artificial ordering requirements due to memory limitations. Workloads that can immediately process and discard data from freeblock requests tend to be able to request more of their needed data at once.

To clarify the types of tasks that fit the freeblock scheduling model, Table 1 presents a sample interface for a freeblock scheduling subsystem, ignoring component and protection boundary issues. This interface is meant to be illustrative only; a comprehensive API would need to address memory allocation, protection, and other issues.

This sample freeblock API has four important characteristics. First, no call into the freeblock scheduling subsystem waits for a disk access. Instead, calls to register requests return immediately, and subsequent callbacks report request completions. This allows applications to register large sets of freeblock

Function Name	Arguments	Description
freeblock_readblocks	*diskaddrs, blksize, callback*	Register freeblock read request(s)
freeblock_writeblocks	*diskaddrs, blksize, buffers, callback*	Register freeblock write request(s)
freeblock_abort	*diskaddrs, blksize*	Abort registered freeblock request(s)
freeblock_promote	*diskaddrs, blksize*	Promote registered freeblock request(s)
**(callback)*	*diskaddr, blksize, buffer*	Call back to task with desired block

Table 1: **A simple interface to a freeblock subsystem.** *freeblock_readblocks* and *freeblock_writeblocks* register one or more single-block freeblock requests, with an application-defined block size. *freeblock_abort* and *freeblock_promote* are applied to previously registered requests, to either cancel pending freeblock requests or convert them to foreground requests. When promoted, multiple contiguous freeblock requests can be merged into a single foreground request. **(callback)* is called by the freeblock subsystem to report availability (or write completion) of a single previously-requested block. When the request was a read, *buffer* points to a buffer containing the desired data. The freeblock subsystem reclaims this buffer when **(callback)* returns, meaning that the callee must either process the data immediately or copy it to another location before returning control.

requests. Second, block sizes are provided with each freeblock request, allowing applications to ensure that useful units are provided to them. Third, freeblock read requests do not specify memory locations for read data. Completion callbacks provide pointers to buffers owned by the freeblock scheduling subsystem and indicate which requested data blocks are in them. This allows tasks to register many more freeblock reads than their memory resources would otherwise allow, giving greater flexibility to the freeblock scheduling subsystem. For example, the data mining example in Section 6 starts by registering freeblock reads for all blocks on the disk. Fourth, freeblock requests can be aborted or promoted to foreground requests at any time. The former allows tasks to register for more data than are absolutely required (e.g., a search that only needs one match). The latter allows tasks to increase the priority of freeblock requests that may soon impact foreground task performance (e.g., a space compression task that has not made sufficient progress).

2.2 Applications

Freeblock scheduling is a new tool, and we expect that system designers will find many unanticipated uses for it. This section describes some of the applications we see for its use.

Scanning applications. In many systems, there are a variety of support tasks that scan large portions of disk contents. Such activities are of direct benefit to users, although they may not be the highest priority of the system. Examples of such tasks include report generation, RAID scrubbing, virus detection, tamper detection [33], and backup. Section 6 explores data mining of an active transaction processing system as a concrete example of such use of free bandwidth.

These disk-scanning application tasks are ideal candidates for free bandwidth utilization. Appropriately structured, they can exhibit all four of the desirable characteristics discussed above. For example, report generation tasks (and data mining in general) often consist of collecting statistics about large sets of small, independent records. These tasks may be of lower priority than foreground transactions, access a large set of blocks, involve no ordering requirements, and process records immediately. Similarly, virus detectors examine large sets of files for known patterns. The files can be examined in any order, though internal statistics for partially-checked files may have significant memory requirements when pieces of files are read in no particular order. Backup applications can be based on physical format, allowing flexible block ordering with appropriate indices, though single-file restoration is often less efficient [28, 14]. Least flexible of these examples would be tamper detection that compares current versions of data to "safe" versions. While the comparisons can be performed in any order, both versions of a particular datum must be available in memory to complete a comparison. Memory limitations are unlikely to allow arbitrary flexibility in this case.

Internal storage optimization. Another promising use for free bandwidth is internal storage system optimization. Many techniques have been developed for reorganizing stored data to improve performance of future accesses. Examples include placing related data contiguously for sequential disk access [37, 57], placing hot data near the center of the disk [56, 48, 3], and replicating data on disk to provide quicker-to-access options for subsequent reads [42, 61]. Other examples include index reorganization [29, 23] and compression of cold data [11]. Section 5 explores segment cleaning in log-structured file systems as a concrete example of such use of free

bandwidth.

Although internal storage optimization activities exhibit the first two qualities listed in Section 2.1, they can impose some ordering and memory restrictions on media accesses. For example, reorganization generally requires clearing (i.e., reading or moving) destination regions before different data can be written. Also, after opportunistically reading data for reorganization, the task must write this data to their new locations. Eventually, progress will be limited by the rate at which these writes can be completed, since available memory resources for buffering such data are finite.

Prefetching and Prewriting. Another use of free bandwidth is for anticipatory disk activities such as prefetching and prewriting. Prefetching is well-understood to offer significant performance enhancements [44, 9, 25, 36, 54]. Free bandwidth prefetching should increase performance further by avoiding interference with foreground requests and by minimizing the opportunity cost of aggressive predictions. As one example, the sequence shown in Figure 1(b) shows one way that the prefetching common in disk firmware could be extended with free bandwidth. Still, the amount of prefetched data is necessarily limited by the amount of memory available for caching, restricting the number of freeblock requests that can be issued.

Prewriting is the same concept in reverse. That is, prewriting is early writing out of dirty blocks under the assumption that they will not be overwritten or deleted before write-back is actually necessary. As with prefetching, the value of prewriting and its relationship with non-volatile memory are well-known [4, 10, 6, 23]. Free bandwidth prewriting has the same basic benefits and limitations as free prefetching.

3 Availability of Free Bandwidth

This section quantifies the availability of potential free bandwidth, which is equal to a disk's total potential bandwidth multiplied by the fraction of time it spends on rotational latency delays. The amount of rotational latency depends on a number of disk, workload, and scheduling algorithm characteristics.

The experimental data in this section was generated with the DiskSim simulator [21], which has been shown to accurately model several modern disk drives [17], including those explored here. By default, the experiments use a Quantum Atlas 10K disk drive and a synthetic workload referred to as *random*. This random workload consists of 10,000 foreground requests issued one at a time with no idle time between requests (closed system arrival model with no think time). Other default parameters for the random workload are request size of 4KB, uniform distribution of starting locations across the disk capacity, and 2:1 ratio of reads to writes.

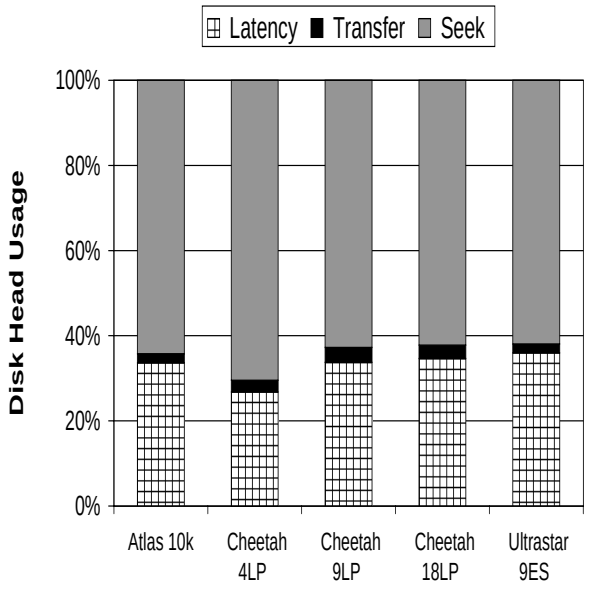


Figure 2: **Disk head usage for several modern disks.** The cross-hatch portion, representing rotational latency, indicates the percentage of total disk bandwidth available as potential free bandwidth.

Most of the bar graphs presented in this section have a common structure. Each bar breaks down disk head usage into several regions that add up to 100%, with each region representing the percentage of the total attributed to the corresponding activity. All such bars include regions for foreground seek times, rotational latencies, and media transfers. The rotational latency region represents the potential free bandwidth (as a percentage of the disk's total bandwidth) available for the disk-workload combination.

3.1 Impact of disk characteristics

Figure 2 shows breakdowns of disk head usage for five modern disk drives whose basic characteristics are given in Table 2. Overall, for the random workload, about one third (27–36%) of each disk's head usage can be attributed to rotational latency. Thus, about one third of the media bandwidth is available for freeblock scheduling, even with no inter-

	Quantum Atlas 10K	Seagate Cheetah 4LP	Seagate Cheetah 9LP	Seagate Cheetah 18LP	IBM Ultrastar 9ES
Year	1999	1996	1997	1998	1998
Capacity	9 GB	4.5 GB	9 GB	9 GB	9 GB
Cylinders	10042	6581	6962	9772	11474
Tracks per cylinder	6	8	12	6	5
Sectors per track	229–334	131–195	167–254	252–360	247–390
Spindle speed (RPM)	10025	10033	10025	10025	7200
Average seek	5.0 ms	7.7 ms	5.4 ms	5.2 ms	7.0 ms
Min-Max seeks	1.2–10.8 ms	0.6–16.1 ms	0.8–10.6 ms	0.7–10.8 ms	1.1–12.7 ms

Table 2: **Basic characteristics of several modern disk drives.**

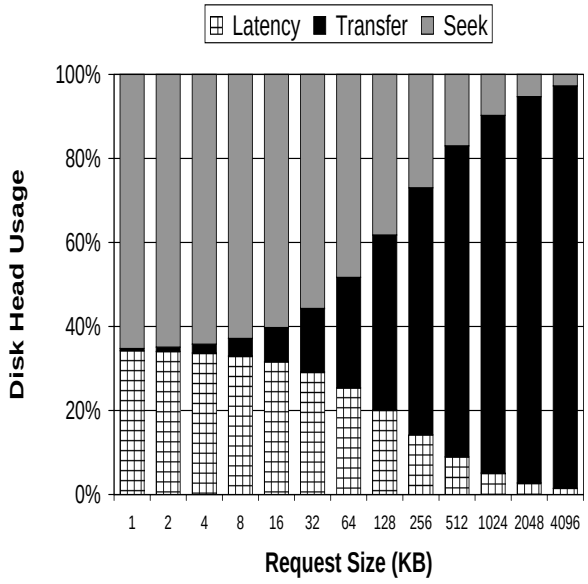


Figure 3: **Disk head usage as a function of request size.**

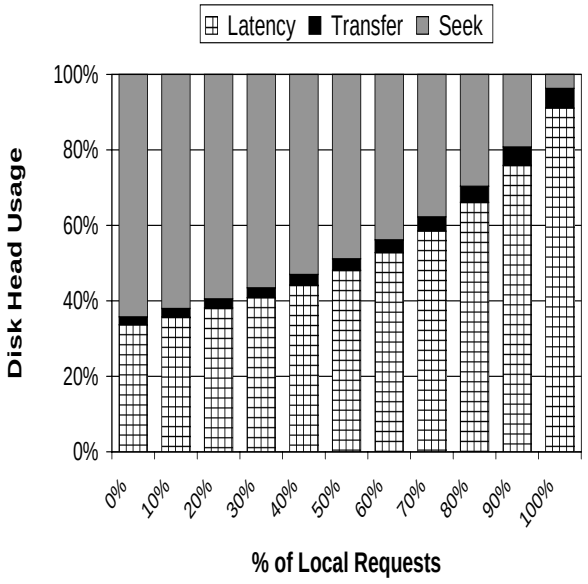


Figure 4: **Disk head usage as a function of access locality.** The default workload was modified such that a percentage of request starting locations are "local" (taken from a normal distribution centered on the last requested location, with a standard deviation of 4MB). The remaining requests are uniformly distributed across the disk's capacity. This locality model crudely approximates the effect of "cylinder group" layouts [38] on file system workloads.

request locality. At a more detailed level, the effect of key disk characteristics can be seen in the breakdowns. For example, the faster seeks of the Cheetah 9LP, relative to the Cheetah 4LP, can be seen in the smaller seek component.

3.2 Impact of workload characteristics

Figure 3 shows how the breakdown of disk head usage changes as the request size of the random workload increases. As expected, larger request sizes yield larger media transfer components, reducing the seek and latency components by amortizing larger transfers over each positioning step. Still, even for large random requests (e.g., 256KB), disk head utilization is less than 55% and potential free bandwidth is 15%.

Figure 4 shows how the breakdown of disk head usage changes as the degree of access locality increases. Because access locality tends to reduce seek distances without directly affecting the other components, this graph shows that both the transfer and latency components increase. For example, when 70% of the requests are within the same "cylinder group" [38] as the last request, almost 60% of the disk head's time is spent in rotational latency and is thus available as free bandwidth. Since disk access locality is a common attribute of many environments, one can generally expect more potential free

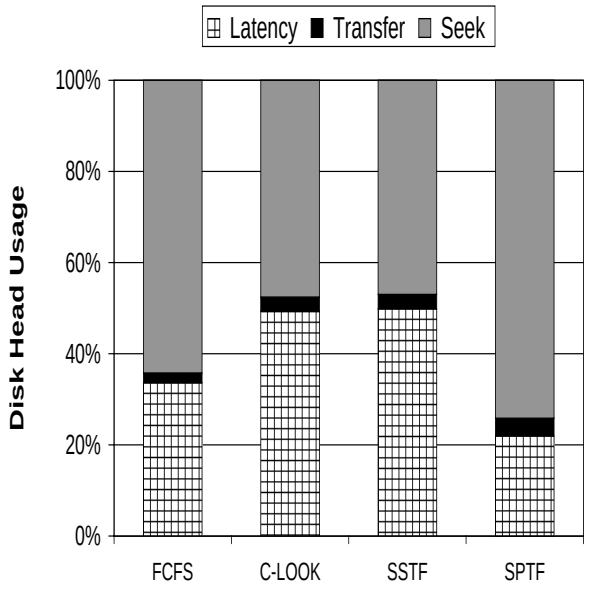


Figure 5: **Disk head usage for several foreground scheduling algorithms.** The default workload was modified to always have 20 requests outstanding. Lowering the number of outstanding requests reduces the differences between the scheduling algorithms, as they all converge on FCFS.

bandwidth than the 33% predicted for the *random* workload.

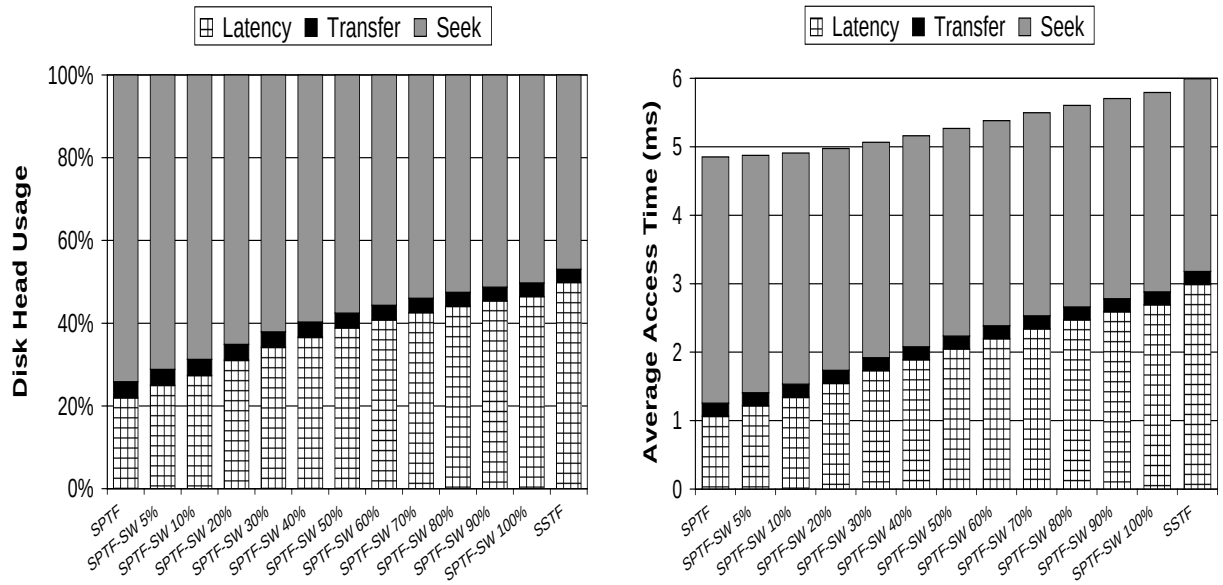
Figure 4 does not show the downside (for freeblock scheduling) of high degrees of locality — starvation of distant freeblock requests. That is, if foreground requests keep the disk head in one part of the disk, it becomes difficult for a freeblock scheduler to successfully make progress on freeblock requests in distant parts of the disk. This effect is taken into account in the experiments of Sections 5 and 6.

3.3 Impact of scheduling algorithm

Figure 5 shows how the breakdown of disk head usage changes for different scheduling algorithms applied to foreground requests. Specifically, four scheduling algorithms are shown: First-Come-First-Served (FCFS), Circular-LOOK (C-LOOK), Shortest-Seek-Time-First (SSTF), and Shortest-Positioning-Time-First (SPTF). FCFS serves requests in arrival order. C-LOOK [49] selects the next request in ascending starting address order; if none exists, it selects the request with the lowest starting address. SSTF [16] selects the request that will incur the shortest seek. SPTF [30, 51, 60] selects the request that will incur the smallest overall positioning delay (seek time plus rotational latency). On average, C-LOOK and SSTF reduce seek times without affecting transfer times and rotational latencies. Therefore, we expect (and observe) the seek component to decrease and the other two to increase. In fact, for this workload, the rotational latency component increases to 50% of the disk head usage. On the other hand, SPTF tends to decrease both overhead components, and Figure 5 shows that the rotational latency component decreases significantly (to 22%) relative to the other scheduling algorithms.

SPTF requires the same basic time predictions as freeblock scheduling. Therefore, its superior performance will make it a common foreground scheduling algorithm in systems that can support freeblock scheduling, making its effect on potential free bandwidth a concern. To counter this effect, we propose a modified SPTF algorithm that is weighted to select requests with both small total positioning delays and large rotational latency components. The algorithm, here referred to as SPTF-SWn%, selects the request with the smallest seek time component among the pending requests whose positioning times are within n% of the shortest positioning time. So, logically, this algorithm first uses the standard SPTF algorithm to identify the next most efficient request, denoted A, to be scheduled. Then, it makes a second pass to find the pending request, denoted B, that has the smallest seek time while still having a total positioning time within n% of A's. Request B is then selected and scheduled. The actual implementation makes a single pass, and its measured computational overhead is only 2–5% higher than that of SPTF. This algorithm creates a continuum between SPTF (when $n = 0$) and SSTF (when $n = \infty$), and we expect the disk head usage breakdown to reflect this.

Figure 6 shows the breakdown of disk head usage and the average foreground request access time when SPTF-SWn% is used for foreground request scheduling. As expected, different values of n result in a range of options between SPTF and SSTF. As n increases, seek reduction becomes a priority, and the rotational latency component of disk head usage increases. At the same time, average access times increase as total positioning time plays a less dominant role in the decision process. Fortunately, the benefits increase rapidly before experiencing diminishing returns, and the penalties increase slowly before ramping up. So, using SPTF-SW40% as an example, we see that a 6% increase in average access time can provide 66% more potential free bandwidth (i.e., 36% rotational latency for SPTF-SW40% compared to SPTF's 22%). This represents half of the



(a) Disk head usage

(b) Average access time

Figure 6: **Disk head usage and average access time with SPTF-SWn% for foreground scheduling.** The default workload was modified to always have 20 requests outstanding.

free bandwidth difference between SPTF and SSTF at much less than the 25% foreground access time difference.

4 Freeblock Scheduling Decisions

Freeblock scheduling is the process of identifying free bandwidth opportunities and matching them to pending freeblock requests. This section describes and evaluates the computational overhead of the freeblock scheduling algorithm used in our experiments.

Our freeblock scheduler works independently of the foreground scheduler and maintains separate data structures. After the foreground scheduler chooses the next request, B, the freeblock scheduler is invoked. It begins by computing the rotational latency that would be incurred in servicing B; this is the free bandwidth opportunity. This computation requires accurate estimates of disk geometry, current head position, seek times, and rotation speed. The freeblock scheduler then searches its list of pending freeblock requests for the most complete use of this opportunity; that is, our freeblock scheduler greedily schedules freeblock requests within free bandwidth opportunities based on the number of blocks that can be accessed.

Our current freeblock scheduler assumes that the most complete use of a free bandwidth opportunity is the maximal answer to the question, "for each track on the disk, how many desired blocks could be accessed in this opportunity?". For each track, t, answering this question requires computing the extra seek time involved with seeking to t and then seeking to B's track, as compared to seeking directly to B's track. Answering this question also requires determining which disk blocks will pass under the head during the remaining rotational latency time and counting how many of them correspond to pending freeblock requests. Note that no extra seek is required for the source track or for B's track.

Obviously, such an exhaustive search can be extremely time consuming. We prune the search space in several ways. First, the freeblock scheduler skips all tracks for which the number of desired blocks is less than the best value found so far. Second, the freeblock scheduler only considers tracks for which the remaining free bandwidth (after extra seek overheads) is greater than the best value found so far. Third, the freeblock scheduler starts by searching the source and destination cylinders (from the previous and current foreground requests), which yield the best choices whenever they are fully populated, and then searching in ascending order of extra seek time. Combined with the first two pruning steps, this ordered search frequently terminates quickly.

The algorithm described above performs well when there is a large number of pending freeblock requests. For example, when 20–100% of the disk is desired, freeblock scheduling decisions are made in 0-2.5ms on a 550MHz Intel Pentium III, which is much less than average disk access times. For such cases, it should be possible to schedule the next freeblock request in real-time before the current foreground request completes, even with a less-powerful CPU. With greater fragmentation of freeblock requests, the time required for the freeblock scheduler to make a decision rises significantly. The worst-case computation time of this algorithm occurs when there are large numbers of small requests evenly distributed across all cylinders. In this case, the algorithm searches a large percentage of the available disk space in the hopes of finding a larger section of blocks than it has already found. To address this problem, one can simply halt searches after some amount of time (e.g., the time available before the previous foreground request completes). In most cases, this has a negligible effect on the achieved free bandwidth. For all experiments in this paper, the freeblock scheduling algorithm was only allowed to search for the next freeblock request in the time that the current foreground request was being serviced.

The base algorithm described here enables significant use of free bandwidth, as shown in subsequent sections. Nonetheless, development of more efficient and more effective freeblock scheduling algorithms is an important area for further work. This will include using both free bandwidth and idle time for background tasks; the algorithm above and all experiments in this paper use only free bandwidth.

5 Free cleaning of LFS segments

The log-structured file system [47] (LFS) was designed to reduce the cost of disk writes. Towards this end, it remaps all new versions of data into large, contiguous regions called segments. Each segment is written to disk with a single I/O operation, amortizing the cost of a single seek and rotational delay over a write of a large number of blocks. A significant challenge for LFS is ensuring that empty segments are always available for new data. LFS answers this challenge with an internal defragmentation operation called *cleaning*. Ideally, all necessary cleaning would be completed during idle time, but this is not always possible in a busy system. The potential and actual penalties associated with cleaning have been the subject of heated debate [50] and several research efforts [52, 37, 7, 39, 59]. With freeblock scheduling, the cost of segment cleaning can be close to zero for many workloads.

5.1 Design

Cleaning of a previously written segment involves identifying the subset of live blocks, reading them into memory, and writing them into the next segment. Live blocks are those that have not been overwritten or deleted by later operations; they can be identified by examining the on-disk segment summary structure to determine the original identity of each block (e.g., block 4 of file 3) and then examining the auxiliary structure for the block's original owner (e.g., file 3's i-node). Segment summaries, auxiliary structures, and live blocks can be read via freeblock requests. There are ordering requirements among these, but live blocks can be read in any order and moved into their new locations immediately.

Like other background LFS cleaners, our freeblock segment cleaner is invoked when the number of empty segments drops below a certain threshold. When invoked, the freeblock cleaner selects several non-empty segments and uses freeblock requests to clean them in parallel with other foreground requests. Cleaning several segments in parallel provides more requests and greater flexibility to the freeblock scheduler. If the freeblock cleaner is not effective enough, the foreground cleaner will be activated when the minimum threshold of free segments is reached.

As live blocks in targeted segments are fetched, they are copied into the in-memory segment that is currently being constructed by LFS writes. Because the live blocks are written into the same segment as data of foreground LFS requests, this method of cleaning is not entirely for free. The auxiliary data structure (e.g., i-node) that marks the location of the block is updated to point to the block's new location in the new segment. When all live blocks are cleaned from a segment on the disk, that segment becomes available for subsequent use.

5.2 Experimental Setup

To experiment with freeblock cleaning, we have modified a log-structured logical disk, called LLD [15]. LLD uses segments consisting of 128 4KB blocks, of which 127 blocks are used for data and one block is used for segment summary. The default implementation of LLD invokes its cleaner only when the number of free segments drops below a threshold (set to two segments). It does not

implement background cleaning. Thus, all segment cleaning activity interferes with the foreground disk I/O. We replaced LLD's default segment selection algorithm for cleaning with Sprite LFS's cost-benefit algorithm[47], yielding better performance for all of the cleaners.

Our experiments were run under Linux 2.2.14 with a combination of real processing times and simulated I/O times provided by DiskSim. To accomplish this, we merged LLD with DiskSim. Computation times between disk I/Os are measured with `gettimeofday`, which uses the Pentium cycle counter. These computation times are used to advance simulation time in DiskSim. DiskSim callbacks report request completions, which are forwarded into the LLD code as interrupts. The contents of the simulated disk are stored in a regular file, and the time required to access this file is excluded from the reported results.

All experiments were run on a 550 MHz Intel Pentium III machine with 256MB of memory. DiskSim was configured to model a modified Quantum Atlas 10K disk. Specifically, since the maximal size of an LLD disk is 400MB, we modified the Atlas 10K specifications to have only one data surface, resulting in a capacity of 1.5GB. Thus, the LLD "partition" occupies about 1/4 of the disk.

To assess the effectiveness of the freeblock cleaner, we used the Postmark v. 1.11 benchmark, which simulates the small-file activity predominant on busy Internet servers [31]. Postmark initially creates a pool of files, then performs a series of transactions, and finally deletes all files created during the benchmark run. A single transaction is one access to an existing file (i.e., read or append) and one file manipulation (i.e., file creation or deletion). We used the following parameter values: 5–10KB file size (default Postmark value), 25000 transactions, and 100 subdirectories. The ratios of read-to-write and create-to-delete were kept at their default values of 1:1. The number of files in the initial pool was varied to provide a range of file system capacity utilizations.

To age the file system, we run the transaction phase twice and report measurements for only the second iteration. The rationale for running the set of transactions the first time is to spread the blocks of the file system among the segments in order to more closely resemble steady-state operation. Recall that Postmark first creates all files before doing transactions which results in all segments being either completely full or completely empty — a situation very unlikely in normal operation.

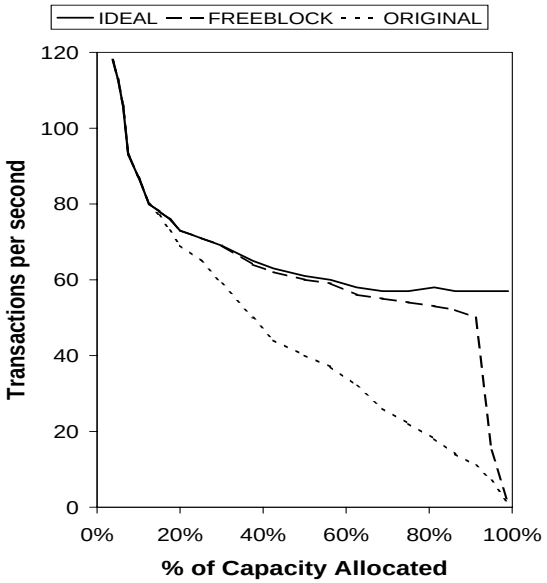


Figure 7: **LLD performance for three cleaning strategies.** Even with a heavy foreground workload (Postmark), segment cleaning can be completed with just freeblock requests until the file system is 93% full.

5.3 Results

Figure 7 shows Postmark's performance for three different cleaner configurations: ORIGINAL is the default LLD cleaner with the Sprite LFS segment selection algorithm. FREEBLOCK is the freeblock cleaner, in which cleaning reads are freeblock requests and cleaning writes are foreground requests. IDEAL subtracts all cleaning costs from ORIGINAL and computes the corresponding throughput, which is unrealistic because infinitely fast foreground cleaning is not possible.

Figure 7 shows the transactions per second for different file system space utilizations, corresponding to different numbers of files initially created by Postmark. The high throughput for low utilizations (less than 8% of capacity) is due to the LLD buffer cache, which absorbs all of the disk activity. IDEAL's performance decreases as capacity utilization increases, because the larger set of files results in fewer cache hits for Postmark's random file accesses. As disk utilization increases, ORIGINAL's throughput decreases consistently due to cleaning overheads, halving performance at 60% capacity and quartering it at 85%. FREEBLOCK maintains performance close to IDEAL (up to 93% utilization). After 93%, there is insufficient time for freeblock cleaning to keep up with the heavy foreground workload, and the performance of FREEBLOCK degrades as the foreground cleaner increasingly dominates performance.

FREEBLOCK's slow divergence from IDEAL between 40% and 93% occurs because FREEBLOCK is being charged for the write cost of cleaned segments while IDEAL is not.

6 Free data mining on OLTP systems

The use of data mining to identify patterns in large databases is becoming increasingly popular over a wide range of application domains and datasets [19, 12, 58]. One of the major obstacles to starting a data mining project within an organization is the high initial cost of purchasing the necessary hardware. Specifically, the most common strategy for data mining on a set of transaction data is to purchase a second database system, copy the transaction records from the OLTP system to the second system each evening, and perform mining tasks only on the second system. This strategy can double capital and operating expenses. It also requires that a company gamble a sizable up-front investment to test suspicions that there may be interesting "nuggets" to be mined from their OLTP databases. With freeblock scheduling, significant mining bandwidth can be extracted from the original system without affecting the original transaction processing activity [45].

6.1 Design

Data mining involves examining large sets of records for statistical features and correlations. Many data mining operations, including nearest neighbor search, association rules [2], ratio and singular value decomposition [34], and clustering [62, 26], eventually translate into a few scans of the entire dataset. Further, individual records can be processed immediately and in any order, matching three of the criteria of appropriate free bandwidth uses.

Our freeblock mining example issues a single freeblock read request for each scan. This freeblock request asks for the entire contents of the database in page-sized chunks. The freeblock scheduler ensures that only blocks of the specified size are provided and that all the blocks requested are read exactly once. However, the order in which the blocks are read will be an artifact of the pattern of foreground OLTP requests.

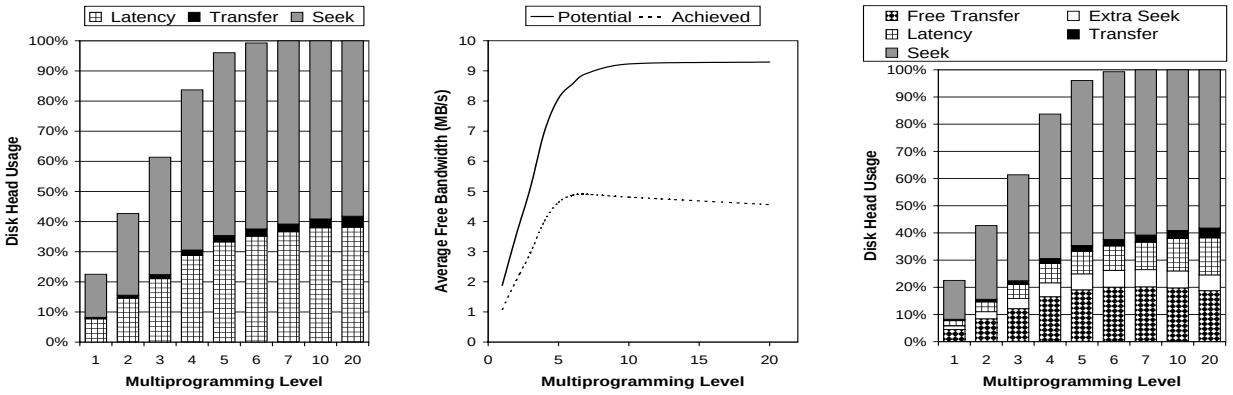
Interestingly, this same design is appropriate for some other storage activities. For example, RAID scrubbing consists of verifying that each disk sector can be read successfully (i.e., that no sector has fallen victim to media corruption). Also, a physical backup consists of reading all disk sectors so that they can be written to another device. The free bandwidth achieved for such scanning activities would match that shown for freeblock data mining in this section.

6.2 Experimental Setup

The experiments in Section 6.3 were conducted using the DiskSim simulator configured to model the Quantum Atlas 10K and a synthetic foreground workload based on approximations of observed OLTP workload characteristics. The synthetic workload models a closed system with per-task disk requests separated by think times of 30 milliseconds. We vary the multiprogramming level (MPL), or number of tasks, of the OLTP workload to create increasing foreground load on the system. For example, a multiprogramming level of ten means that there are ten requests active in the system at any given point, either queued at the disk or waiting in think time. The OLTP requests are uniformly-distributed across the disk's capacity with a read to write ratio of 2:1 and a request size that is a multiple of 4 kilobytes chosen from an exponential distribution with a mean of 8 kilobytes. Validation experiments (in [45]) show that this workload is sufficiently similar to disk traces of Microsoft's SQL server running TPC-C for the overall freeblock-related insights to apply to more realistic OLTP environments. The background data mining workload uses free bandwidth to make full scans of the disk's contents in 4 KB blocks, completing one scan before starting the next. All simulations run for the time required for the background data mining workload to complete ten full disk scans, and the results presented are averages across these ten scans. The experiments ignore bus bandwidth and record processing overheads, assuming that media scan times dominate; this assumption might be appropriate if the mining data is delivered over distinct buses to dedicated processors either on a small mining system or in Active Disks.

6.3 Results

Figure 8 shows the disk head usage for the foreground OLTP workload at a range of MPLs and the free bandwidth achieved by the data mining task. Low OLTP loads result in low data mining throughput, because little potential free bandwidth exists when there are few foreground requests. Instead, there is a significant amount of idle disk head time that could be used for freeblock requests, albeit not without some effect on foreground response times.



(a) OLTP disk head usage

(b) Free mining bandwidth

(c) Combined head usage

Figure 8: **Average freeblock-based data mining performance.** (a) shows the disk head usage breakdown for the foreground OLTP workload at various MPLs. (b) shows the overall free bandwidth delivered to the data mining application for the same points. (c) shows the disk head usage breakdown with both the foreground OLTP workload and the background data mining application.

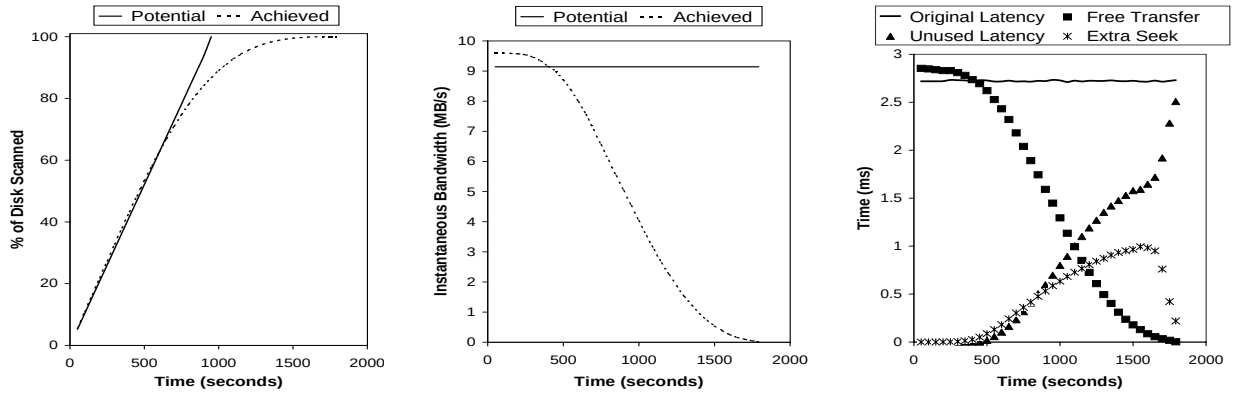
Our study here focuses strictly on use of free bandwidth. As the foreground load increases, opportunities to service freeblock requests are more plentiful, increasing data mining throughput to about 4.9 MB/s (21% of the Atlas 10K's 23MB/s full potential bandwidth). This represents a 7× increase in useful disk head utilization, from 3% to 24%, and it allows the data mining application to complete over 47 full "scans per day" [24] of this 9GB disk with no effect on foreground OLTP performance.

However, as shown in Figure 8b, freeblock scheduling realizes only half of the potential free bandwidth for this environment. As shown in Figure 8c, 18% of the remaining potential is lost to extra seek time, which occurs when pending freeblock requests only exist on a third track (other than the previous and current foreground request). The remaining 28% continues to be rotational latency, either as part of freeblock requests or because no freeblock request could be serviced within the available slot.

Figure 9 helps to explain why only half of the potential free bandwidth is realized for data mining. Specifically, it shows data mining progress and per-OLTP-request breakdown as functions of the time spent on a given disk scan. The main insight here is that the efficiency of freeblock scheduling (i.e., achieved free bandwidth divided by potential free bandwidth) drops steadily as the set of still-desired background blocks shrinks. As the freeblock scheduler has more difficulty finding conveniently-located freeblock requests, it must look further and further from the previous and current foreground requests. As shown in Figure 9c, this causes extra seek times to increase. Unused rotational latency also increases as freeblock requests begin to incur some latency and as increasing numbers of foreground rotational latencies are found to be too small to allow any pending freeblock request to be serviced. As a result, servicing the last few freeblock requests of a full scan takes a long time; for example, the last 5% of the freeblock requests take 30% of the total time for a scan.

One solution to this problem would be to increase the priority of the last few freeblock requests, with a corresponding impact on foreground requests. The challenge would be to find an appropriate trade-off between impact on the foreground and improved background performance.

An alternate solution would be to take advantage of the statistical nature of many data mining queries. Statistical sampling has been shown to provide accurate results for many queries and internal database operations after accessing only a (randomly-selected) subset of the total dataset [43, 13]. Figure 10 shows the impact of such statistical data mining as a function of the percentage of the dataset needed; that is, the freeblock request is aborted when enough of the dataset has been mined. Assuming that freeblock scheduling within the foreground OLTP workload results in sufficiently random data selection or that the sampling algorithm is adaptive to sampling biases [13], sampling can significantly increase freeblock scheduler efficiency. When any 95% of the dataset is sufficient,



(a) Percent complete (b) Instantaneous free BW (c) Per-request breakdown

Figure 9: **Freeblock-based data mining progress for MPL 7.** (a) shows the potential and achieved scan progress. (b) shows the corresponding instantaneous free bandwidth. (c) shows the usage of potential free bandwidth (i.e., original OLTP rotational latency), partitioning it into free transfer time, extra seek time, and unused latency. As expected, the shape of the Free Transfer line in (c) matches that of the achieved instantaneous mining bandwidth in (b). Both exceed the potential free bandwidth early in the scan because many foreground OLTP transfers can also be used by the freeblock scheduler for mining requests when most blocks are still needed.

efficiency is 40% higher than for full disk scans. For 80% of the dataset, efficiency is at 90% and data mining queries can complete over 90 samples of the dataset per day.

7 Related Work

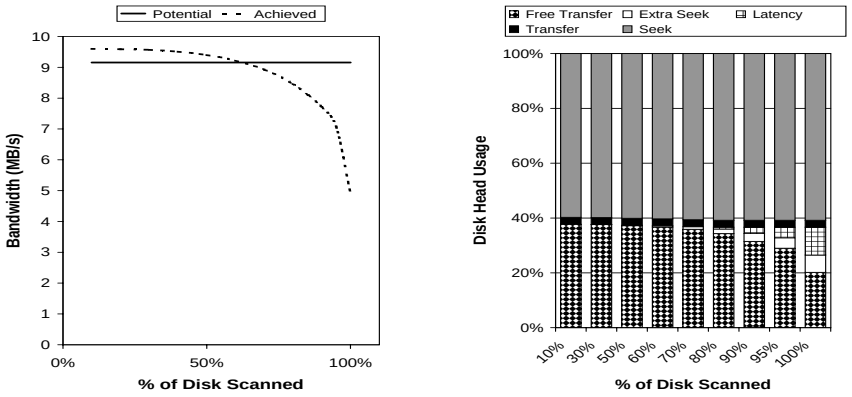
System designers have long struggled with disk performance, developing many approaches to reduce mechanical positioning overheads and to amortize these overheads over large media transfers. When effective, all of these approaches increase disk head utilization for foreground workloads and thereby reduce the need for and benefits of freeblock scheduling; none have yet eliminated disk performance as a problem. The remainder of this section discusses work specifically related to extraction and use of free bandwidth.

The characteristics of background workloads that can most easily utilize free bandwidth are much like those that can be expressed well with dynamic set [55] and disk-directed I/O [35] interfaces. Specifically, these interfaces were devised to allow application writers to expose order-independent access patterns to storage systems. Application-hinted prefetching interfaces [9, 44] share some of these same qualities. Such interfaces may also be appropriate for specifying background activities to freeblock schedulers.

Use of idle time to handle background activities is a long-standing practice in computer systems. A subset of the many examples, together with a taxonomy of idle time detection algorithms, can be found in [23]. Freeblock scheduling complements exploitation of idle time. It also enjoys two superior qualities: (1) ability to make forward progress during busy periods and (2) ability to make progress with **no** impact on foreground disk access times. Starting a disk request during idle time can increase the response time of subsequent foreground requests, by making them wait or by moving the disk head.

In their exploration of write caching policies, Biswas, et al., evaluate a free prewriting mechanism called *piggybacking* [6]. Although piggybacking only considers blocks on the destination track or cylinder, they found that most write-backs could be completed for free across a range of workloads and cache sizes. Relative to their work, our work generalizes both the freeblock scheduling algorithm and the uses for free bandwidth.

Freeblock scheduling relies heavily on the ability to accurately predict mechanical positioning delays (both seek times and rotational latencies). The firmware of most high-end disk drives now supports Shortest-Positioning-Time-First (SPTF) scheduling algorithms, which require similar predictions. Based on this fact, we are confident that freeblock scheduling is feasible. However, it remains to be seen whether freeblock scheduling can be effective outside



(a) Free mining bandwidth

(b) Disk head usage

Figure 10: **Freeblock-based data mining performance for statistical queries.** Here, it is assumed that any X% of the disk's data satisfy the needs of a query scan. Below 60%, achieved free bandwidth exceeds potential free bandwidth because of the ability to satisfy freeblock requests from foreground transfers

of disk drive firmware, where complete knowledge of current state and internal algorithms is available.

Freeblock scheduling resembles advanced disk schedulers for environments with a mixed workload of real-time and non-real-time activities. While early real-time disk schedulers gave strict priority to real-time requests, more recent schedulers try to use slack in deadlines to service non-real-time requests without causing the deadlines to be missed [53, 41, 5, 8]. Freeblock scheduling relates to conventional priority-based disk scheduling (e.g., [10, 20]) roughly as modern real-time schedulers relate to their predecessors. However, since non-real-time requests have no notion of deadline slack, freeblock scheduling must be able to service background requests without extending the access latencies of foreground requests at all. Previous disk scheduler architectures would not do this well for non-periodic foreground workloads, such as those explored in this paper.

While freeblock scheduling can provide free media bandwidth, use of such bandwidth also requires some CPU, memory, and bus resources. One approach to addressing these needs is to augment disk drives with extra resources and extend disk firmware with application-specific functionality [1, 32, 46]. Potentially, such resources could turn free bandwidth into free functionality; Riedel, et al., [45] argue exactly this case for the data mining example of Section 6.

Another interesting use of accurate access time predictions and layout information is *eager writing*, or remapping new versions of disk blocks to free locations very near the disk head [27, 18, 40, 57]. We believe that eager writing and freeblock scheduling are strongly complementary concepts. Although eager writing decreases available free bandwidth during writes by eliminating many seek and rotational delays, it does not do so for reads. Further, eager writing could be combined with freeblock scheduling when using a write-back cache. Finally, as with the LFS cleaning example in Section 5, free bandwidth represents an excellent resource for cleaning and reorganization enhancements to the base eager writing approach [57].

8 Conclusions

This paper describes freeblock scheduling, quantifies its potential under various conditions, and demonstrates its value for two specific application environments. By servicing background requests in the context of mechanical positioning for normal foreground requests, 20–50% of a disk's potential media bandwidth can be obtained with no impact on the original requests. Using simulation, this paper shows that this free bandwidth can be used to clean LFS segments on busy file servers and to mine data on active transaction processing systems.

These results indicate significant promise, but additional experience is needed to refine and realize freeblock scheduling in practice. For example, it remains to be seen whether freeblock scheduling can be implemented outside of modern disk drives, given their high-level interfaces and complex firmware algorithms. Even inside disk firmware, freeblock scheduling will need to conservatively deal with seek

and settle time variability, which may reduce its effectiveness. More advanced freeblock scheduling algorithms will also be needed to deal with request fragmentation, starvation, and priority mixes.

Acknowledgments

We thank Peter Druschel, Garth Gibson, Andy Klosterman, David Petrou, Jay Wylie, Ted Wong and the anonymous reviewers for helping us to refine this paper. We thank the members and companies of the Parallel Data Consortium (including CLARiiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. We also thank IBM Corporation for supporting our research efforts. This work was also partially funded by the National Science Foundation via CMU's Data Storage Systems Center.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, California), pages 81–91. ACM, 3–7 October 1998.

[2] R. Agrawal and J. Schafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, **8**(6), December 1996.

[3] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, **13**(2):89–121, May 1995.

[4] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):10–22, October 1992.

[5] Paul R. Barham. A Fresh Approach to File System Quality of Service. *IEEE 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1997.

[6] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, May 1993.

[7] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans), pages 277–288. Usenix Association, 16–20 January 1995.

[8] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk Scheduling with Quality of Service Guarantees. *IEEE International Conference on Multimedia Computing and Systems*, 1999.

[9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, **14**(4):311–343, November 1996.

[10] Scott C. Carson and Sanjeev Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, **18**(1):44–54, January 1992.

[11] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA 8–11 April 1991), pages 200–211. ACM, 1991.

[12] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, **26**(1):55, 1997.

[13] S. Chaudhuri, R. Motwani, and V. Narasayya. Random Sampling for Histogram Construction: How much is enough? *SIGMOD Record*, **27**(2):436-447, 1998.

[14] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference*, March 1998.

[15] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: a new approach to improving file systems. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 15–28, 5–8 December 1993.

[16] Peter J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.

[17] Database of validated disk parameters for DiskSim. http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html.

[18] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Winter USENIX Technical Conference* (San Francisco, CA), pages 237–251. Usenix, 20–24 January 1992.

[19] Ussama Fayyad. Taming the Giants and the Monsters: Mining Large Databases for Nuggets of Knowledge. *Databases Programming and Design*, March 1998.

[20] Gregory R. Ganger and Yale N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Transactions Vol. 47 No. 6*, June 1998.

[21] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 1.0 Reference Manual*, CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[22] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[23] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA), pages 201–212. Usenix Association, Berkeley, CA, 16–20 January 1995.

[24] Jim Gray and Goetz Graefe. Storage metrics. Microsoft Research, Draft of March 1997.

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

[26] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *SIGMOD Record*, **27**(2):73, 1998.

[27] Robert B. Hagmann. *Low latency logging.* CSL–91–1. Xerox Palo Alto Research Center, CA, February 1991.

[28] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 239–249. ACM, February 1999.

[29] Erin H. Herrin II and Raphael Finkel. An ASCII database for fast queries of relatively stable data. *Computing Systems*, **4**(2):127–155. Usenix Association, Spring 1991.

[30] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position.* HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[31] Jeffrey Katcher. *PostMark: a new file system benchmark.* TR3022. Network Appliance, October 1997.

[32] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, **27**(3):42–52, September 1998.

[33] Gene H. Kim and Eugene H. Spafford. The design and imple-

mentation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, Virginia), pages 18–29, 2–4 November 1994.

[34] F. Korn, A. Labrinidis, Y. Kotidis, and C. Faloutsos. Ratio Rules: A New Paradigm for Fast, Quantifiable Data Mining. *VLDB*, August 1998.

[35] David Kotz. Disk-directed I/O for MIMD multiprocessors. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 61–74. Usenix Association, 14–17 November 1994.

[36] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. *Hot Topics in Operating Systems* (Rio Rico, Arizona), pages 14–19, 29–30 March 1999.

[37] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[38] 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.

[39] Bruce McNutt. Background data movement in a log-structured disk subsystem. *IBM Journal of Research and Development*, **38**(1):47–58, 1994.

[40] Jai Menon, James Roche, and Jim Kasson. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing*, **17**(1–2):129–139, January-February 1993.

[41] Anastasio Molano, Kanaka Juvva, and Ragunathan Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings Real-Time Systems Symposium* (San Francisco, CA, 2–5 December 1997), pages 155–165. Institute of Electrical and Electronics Engineers Comp. Soc., 1997.

[42] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[43] F. Olken and D. Rotem. Simple random sampling from relational databases. *VLDB*, pages 160-169, 1986.

[44] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5):79–95, 3–6 December 1995.

[45] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD Conference 2000* (Dallas, TX), pages 13–21, 14–19 May 2000.

[46] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24–27 August, 1998). Published as *Proceedings VLDB.*, pages 62–73. Morgan Kaufmann Publishers Inc., 1998.

[47] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[48] C. Ruemmler and J. Wilkes. *Disk Shuffling.* HPL-91-156. Hewlett-Packard Laboratories, Palo Alto, CA, October 1991.

[49] P. H. Seaman, R. A. Lind, and T. L. Wilson. On teleprocessing system design, part IV: an analysis of auxiliary-storage activity. *IBM Systems Journal*, **5**(3):158–170, 1966.

[50] Margo Seltzer. LFS and FFS Supplementary Information, 1995. http://www.eecs.harvard.edu/~margo/usenix.195.

[51] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC), pages 313–323, 22–26 January 1990.

[52] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.

[53] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI). Published as *Performance Evaluation Review*, **26**(1):44–55, June 1998.

[54] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why Does File System Prefetching Work? *USENIX Technical Conference* (Monterey, California, June 6-11 1999), pages 71–83. USENIX, 1999.

[55] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):252–263. ACM, 1997.

[56] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, **20**(3):225–242, March 1990.

[57] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, Winter 1998.

[58] J. Widom. Research Problems in Data Warehousing. *Conference on Information and Knowledge Management*, November 1995.

[59] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[60] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. *Scheduling for modern disk drives and non-random workloads.* CSE–TR–194–94. Department of Computer Science and Engineering, University of Michigan, March 1994.

[61] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading Capacity for Performance in a Disk Array. *Symposium on Operating Systems Design and Implementation*, October 2000.

[62] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery*, **2**(1), 1997.