Freeblock Scheduling Outside of Disk Firmware

Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger
Carnegie Mellon University

Abstract

Freeblock scheduling replaces a disk drive's rotational latency delays with useful background media transfers, potentially allowing background disk I/O to occur with no impact on foreground service times. To do so, a freeblock scheduler must be able to very accurately predict the service time components of any given disk request — the necessary accuracy was not previously considered achievable outside of disk firmware. This paper describes the design and implementation of a working external freeblock scheduler running either as a user-level application atop Linux or inside the FreeBSD kernel. This freeblock scheduler can give 15% of a disk's potential bandwidth (over 3.1MB/s) to a background disk scanning task with almost no impact (less than 2%) on the foreground request response times. This can increase disk bandwidth utilization by over 6×.

1 Introduction

Freeblock scheduling is an exciting new approach to utilizing more of a disk's potential media bandwidth. It consists of anticipating rotational latency delays and filling them with media transfers for background tasks. Via simulation, our prior work [14] indicated that 20–50% of a never-idle disk's bandwidth could be provided to background applications with no effect on foreground response times. This *free bandwidth* was shown to enable free segment cleaning in a busy log-structured file system (LFS), or free disk scans (e.g., for data mining or disk media scrubbing) in an active transaction processing system.

At the time of that writing, we and others believed that freeblock scheduling could only be done effectively from inside the disk's firmware. In particular, we did not believe that sufficient service time prediction accuracy could be achieved from outside the disk. We were wrong.

This paper describes and evaluates working prototypes of freeblock scheduling on Linux and within the FreeBSD kernel. Recent research has successfully demonstrated software-only Shortest-Positioning-Time-First (SPTF) [12, 25] schedulers [28, 31], but their prediction accuracies were not high enough to support freeblock scheduling. To squeeze extra media transfers into rotational latency gaps, a freeblock scheduler must be able to predict access times to within 200–300μs. It must also be able to deal with the drive's cache prefetching algorithms, since the most efficient use of a free bandwidth opportunity is on the same track as a foreground request.

These requirements can be met with two extensions to the common external SPTF design: limited command queueing and request merging. First, by keeping two requests outstanding at all times, an external scheduler can focus on just media access delays; the disk's firmware will overlap bus and command processing overheads for any one request with the media access of another. This tighter focus simplifies the scheduler's timing predictions, allowing it to achieve the necessary accuracy. Second, by merging physically adjacent free bandwidth and foreground fetches into a single request, an external scheduler can employ same-track fetches without confusing the firmware's prefetching algorithms.

With its service time prediction accuracy, our external scheduler's SPTF decisions match those of the disk's firmware, and its freeblock scheduling decisions are effective. On the other hand, the achieved free bandwidth is 35% lower than the earlier simulations, because the external prediction accuracies and control are not perfect. Nonetheless, the goals of freeblock scheduling are met: potential free bandwidth is used for background activities with (almost) no impact on foreground response times. For example, when using free bandwidth to scan the entire disk during on-line transaction processing, we measure 3.1 MB/s of steady-state progress or 37 free scans per day on a 9 GB disk. When employing freeblock scheduling, foreground response times increase by less than 2%.

The remainder of this paper is organized as follows. Section 2 describes freeblock scheduling. Section 3 describes challenges involved with implementing freeblock scheduling outside of disk firmware. Section 4 describes our implementation. Section 5 evaluates our external freeblock scheduler. Section 6 discusses related work. Section 7 summarizes this paper's contributions.

2 Freeblock Scheduling

Current high-end disk drives offer media bandwidths in excess of 40 MB/s, and the recent rate of improvement in media bandwidth exceeds 40% per year. Unfortunately, mechanical positioning delays limit most systems to only 2–15% of the potential media bandwidth. We recently

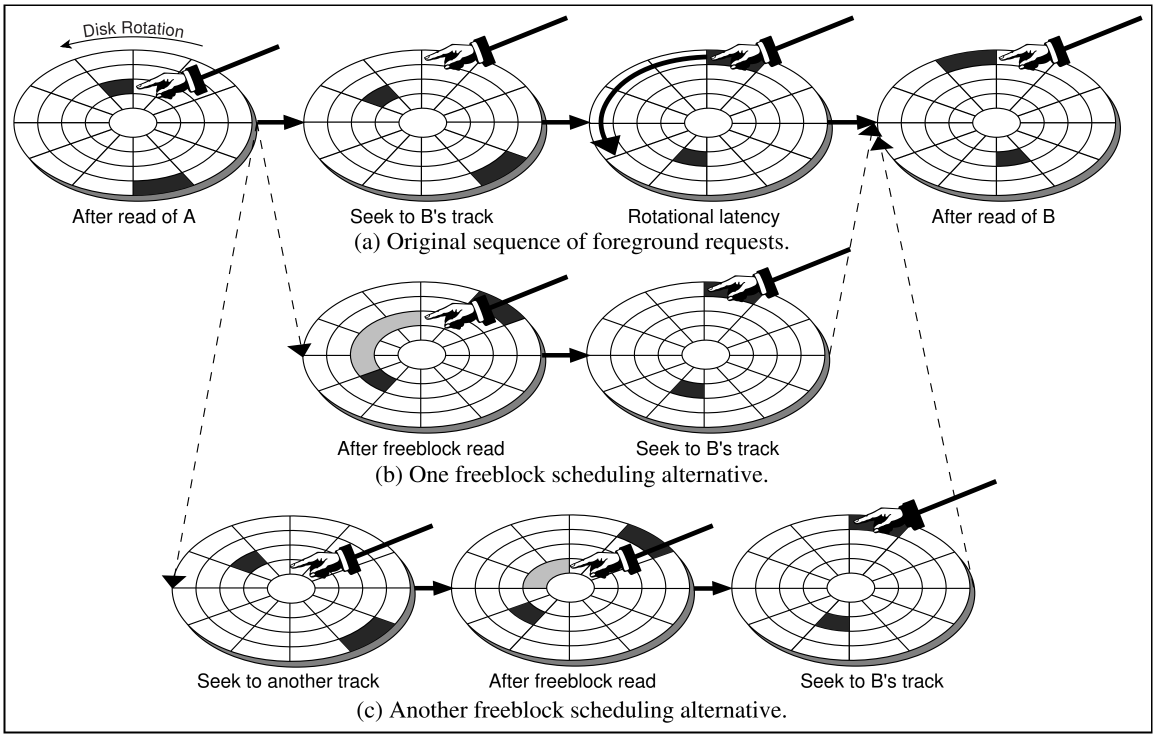


Figure 1: **Illustration of two freeblock scheduling possibilities.** Three sequences of steps are shown, each starting after completing the foreground request to block *A* and finishing after completing the foreground request to block *B*. Each step shows the position of the disk platter, the read/write 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 (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.

proposed freeblock scheduling as an approach to increasing media bandwidth utilization [14, 21]. By interleaving low-priority disk activity with the normal workload (here referred to as background and foreground, respectively), a freeblock scheduler can replace many foreground rotational latency delays with useful background media transfers. With appropriate freeblock scheduling, background tasks can make forward progress without any increase in foreground service times. Thus, the background disk activity is completed for free during the mechanical positioning for foreground requests.

This section describes the free bandwidth concept in greater detail, discusses how it can be used in systems, and outlines how a freeblock scheduler works. Most of the concepts were first described in our prior work [14] and are reviewed here for completeness.

2.1 Where the free bandwidth lives

At a high-level, the time required for a disk media access, T_{access}, can be computed as a sum of seek time, T_{seek}, rotational latency, T_{rotate}, and media access time, $T_{transfer}$:

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

Of T_{access}, only the $T_{transfer}$ component represents useful utilization of the disk head. Unfortunately, the other two components usually dominate. 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. If that time can be predicted, there is an opportunity to do something more useful than just waiting for desired sectors to arrive at the disk head.

Freeblock scheduling is the process of identifying free bandwidth opportunities and matching them to pending background requests. It 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.

The potential free bandwidth in a system is equal to the disk's potential media 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. For random small requests, about 33% of the total time is rotational latency for most disks. This percentage decreases with increasing request size, becoming 15% for 256 KB requests, because more time is spent on data transfer. This percentage increases with increasing locality, up to 60% when 70% of requests are in the most recent "cylinder group" [16], because less time is spent on the shorter seeks. The value is about 50% for seek-reducing scheduling algorithms (e.g., C-LOOK [17, 24] and Shortest-Seek-Time-First [9]) and about 20% for scheduling algorithms that reduce overall positioning time (e.g., Shortest-Positioning-Time-First).

2.2 Uses for 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, and no particular order of access.

These characteristics are common to many disk-intensive background tasks that are designed to occur during otherwise idle time. For example, in many systems, there are a variety of support tasks that scan large portions of disk contents, such as report generation, RAID scrubbing, virus detection, and backup. Another set of examples is the many defragmentation [15, 29] and replication [18, 31] techniques that have been developed to improve the performance of future accesses. A third set of examples is anticipatory disk activities such as prefetching [7, 11, 13, 19, 27] and prewriting [2, 4, 8, 10].

Using simulation, our previous work explored two specific uses of freeblock scheduling. One set of experiments showed that cleaning in a log-structured file system [22] can be done for free even when there is no truly idle time, resulting in up to a 300% increase in application performance. A second set of experiments explored 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 9 GB disk can be made with no impact on OLTP performance. This resulted in a 7× increase in media bandwidth utilization.

2.3 Freeblock scheduling

In a system supporting freeblock scheduling, there are two types of requests: foreground requests and freeblock (background) requests. Foreground requests are the normal workload of the system, and they will receive top priority. Freeblock requests specify the background disk activity for which free bandwidth should be used. As an example, a freeblock request might specify that a range of 100,000 disk blocks be read, but in no particular order — as each block is retrieved, it is handed to the background task, processed immediately, and then discarded. A request of this sort gives the freeblock scheduler the flexibility it needs to effectively utilize free bandwidth opportunities.

Foreground and freeblock requests are kept in separate lists and scheduled separately. The foreground scheduler runs first, deciding which foreground request should be serviced next in the normal fashion. Any conventional scheduling algorithm can be used. Device driver schedulers usually employ seek-reducing algorithms, such as C-LOOK or Shortest-Seek-Time-First. Disk firmware schedulers usually employ Shortest-Positioning-Time-First (SPTF) algorithms [12, 25] to reduce overall positioning overheads (seek time plus rotational latency).

After the next foreground request (request *B* in Figure 1) is determined, the freeblock scheduler computes how much rotational latency would be incurred in servicing *B*; this is the free bandwidth opportunity. Like SPTF, 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 a good match. (Section 4.3 describes a specific freeblock scheduling algorithm.) After making its choice, the scheduler issues any free bandwidth accesses and then request *B*.

3 Fine-grain External Disk Scheduling

Fine-grain disk scheduling algorithms (e.g., Shortest-Positioning-Time-First and freeblock) must accurately predict the time that a request will take to complete. Inside disk firmware, the information needed to make such predictions is readily available. This is not the case outside the disk drive, such as in disk array firmware or OS device drivers.

Modern disk drives are complex systems, with finely-engineered mechanical components and substantial runtime systems. Behind standardized high-level interfaces, disk firmware algorithms map logical block numbers (LBNs) to physical sectors, prefetch and cache data, and schedule media and bus activity. These algorithms vary among disk models, and evolve from one disk generation to the next. External schedulers are isolated from necessary details and control by the same high-level interfaces that allow firmware engineers to advance their algorithms while retaining compatibility. This section outlines major challenges involved with fine-grain external scheduling, the consequences of these challenges, and some solutions that mitigate the negative effects of these consequences.

3.1 Challenges

The challenges faced by a fine-grained external scheduler largely result from disks' high-level interfaces, which hide internal information and restrict external control. Specific challenges include coarse observations, non-constant delays, non-preemption, on-board caching, in-drive scheduling, computation of rotational offsets, and disk-internal activities.

Coarse observations. An external scheduler sees only the total response time for each request. These coarse observations complicate both the scheduler's initial configuration and its runtime operation. During initial configuration, the scheduler must deduce from these observations the individual component delays (e.g., mechanical positioning, data transfer, and command processing) as well as the amount of their overlap. These delays must be well understood for an external scheduler to accurately predict requests' expected response times. During runtime operation, the scheduler must deduce the disk's current state after each request; without this knowledge, the subsequent scheduling decision will be based on inaccurate information.

Non-constant delays. Deducing component delays from coarse observations is made particularly difficult by the inherent inter-request variation of those delays. If the delays were all constant, deduction could be based on solving sets of equations (response time observations) to figure out the unknowns (component delays). Instead, the delays and the amount of their overlap vary. As a result, an external scheduler must deduce moving targets (the component delays) from its coarse observations. In addition, the variation will affect response times of scheduled requests, and so it must be considered in making scheduling decisions. Figure 2 illustrates the effect of variable overlap between bus transfer and media transfer on the observed response time.

Non-preemption. Once a request is issued to the disk, the scheduler cannot change or abort it. The SCSI protocol does include an ABORT message, but most device drivers do not support it and disks do not implement it efficiently. They view it as an unexpected condition, so it is usually more efficient to just allow a request to complete. Thus, an external scheduler must take care in the decisions it makes.

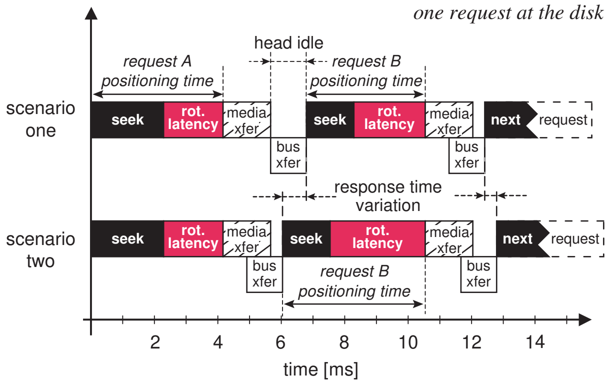


Figure 2: **Effects of uncertainty on prediction accuracy.** This figure shows two possible scenarios of observed response times when employing external scheduling. In each scenario, the scheduler issues request *A*, waits for its completion, and then issues request *B*. The two scenarios only differ in the amount of overlap between the media and bus transfers. The varying overlap has different effects on the positioning time of request B and therefore on the amount of available free bandwidth.

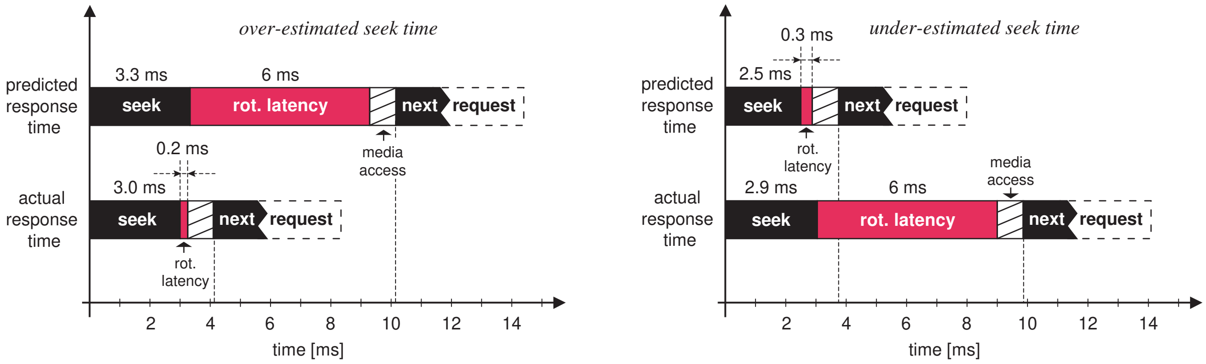
On-board caching. Modern disks have large on-board caches. Exploiting its local knowledge, disk firmware prefetches sectors into this cache based on physical locality. Usually, the prefetching will occur opportunistically during idle time and rotational latency periods[1]. Sometimes, however, the firmware will decide that a sequential read pattern will be better served by delaying foreground requests for further prefetching. An external scheduler is unlikely to know the exact algorithms used for replacement, prefetching, or write-back (if used). As a result, cache hits and prefetch activities will often surprise it.

In-drive scheduling. Modern disks support command queueing, and they internally schedule queued requests to maximize efficiency. An external scheduler that wishes to maintain control must either avoid command queueing or anticipate possible modification of its decisions.

Computation of rotational offsets. A disk's rotation speed may vary slightly over time. As a result, an external scheduler must occasionally resynchronize its understanding of the disk's rotational offset. Also, whenever making a scheduling decision, it must update its view of the current offset.

Internal disk activities. Disk firmware must sometimes execute internal functions (e.g., thermal recalibration) that are independent of any external requests. Unless a

[1]Freeblock scheduling often removes the disk's opportunity to prefetch during rotational latency periods. It does so to fetch known-to-be-wanted data, which we argue is a more valuable activity. In part, we assert this because the lost prefetching will rarely eliminate subsequent media accesses, since the prefetched sectors are usually not forward in LBN order and not aligned to any block boundary or size.



(a) **Seek time over-estimation.** The larger predicted seek of 3.3 ms suggests a full rotation, resulting in a predicted response time of 10.2 ms. Since the actual seek is smaller (3.0 ms), the extra rotation does not occur and the request completes in 4.2 ms, resulting in a -6.0 ms prediction error.

(b) **Seek time under-estimation** The predicted seek of 2.5 ms results in a prediction of rotational latency of 0.3 ms and a predicted response time of 3.8 ms. Since the actual seek is larger (2.9 ms), the disk will suffer an extra rotation resulting in a response time of 9.8 ms. The prediction error is $+6.0$ ms.

Figure 3: **The effects of mispredicted seek times.**

device driver uses recent S.M.A.R.T. interface extensions to avoid these functions, an unexpected internal activity will occasionally invalidate the scheduler's predictions.

3.2 Consequences

The challenges listed above have five main consequences on the operation of an external fine-grained disk scheduler.

Complexity. Both the initial configuration and runtime operation of an external scheduler will be complex and disk-specific. As a result, substantial engineering may be required to achieve robust, effective operation. Worse, effective freeblock scheduling requires very accurate service time predictions to avoid disrupting foreground request performance.

Seek misprediction. When making a scheduling decision, the scheduler predicts the mechanical delays that will be incurred for each request. When there are small errors in the initial configuration of the scheduler or variations in seek times for a given cylinder distance, the scheduler will sometimes mispredict the seek time. When it does, it will also mispredict the rotational latency.

When a scheduler over-estimates a request's seek time (see Figure 3(a)), it may incorrectly decide that the disk head will "just miss" the desired sectors and have to wait almost a full rotation. With such a large predicted delay, the scheduler is unlikely to select this request even though it may actually be the best option.

When the scheduler under-estimates a request's seek time (see Figure 3(b)), it may incorrectly decide that the disk head will arrive just in time to access the desired sectors with almost no rotational latency. Because of the small predicted delay, the scheduler is likely to select this request even though it is probably a bad choice.

Under-estimated seeks can cause substantial unwanted extra rotations for foreground requests. Over-estimated seeks usually do not cause significant problems for foreground scheduling, because selecting the second-best request usually results in only a small penalty. When the foreground scheduler is used in conjunction with a freeblock scheduler, however, an over-estimated seek may cause a freeblock request to be inserted in place of an incorrectly predicted large rotational latency. Like a self-fulfilling prophecy, this will cause an extra rotation before servicing the next foreground request even though it would not otherwise be necessary.

Idle disk head time. The response time for a single request includes mechanical actions, bus transfers, and command processing. As a result, the read/write head can be idle part of the time, even while a request is being serviced. Such idleness occurs most frequently when acquiring and utilizing the bus to transfer data or completion messages. Although an external scheduler can be made to understand such inefficiencies, they can reduce its ability to utilize the potential free bandwidth found in foreground rotational latencies.

Incorrectly-triggered prefetching. Freeblock scheduling works best when it picks up blocks on the source or destination tracks of a foreground seek. However, if the disk observes two sequential READs, it may assume a sequential access pattern and initiate prefetching that causes a delay in handling subsequent requests. If one of these READs is from the freeblock scheduler, the disk will be acting on misinformation since the foreground workload may not be sequential.

Loss of head location information. Several of the challenges will cause an external scheduler to sometimes make decisions based on inaccurate head location information. For example, this will occur for unexpected cache hits, internal disk activity, and triggered foreground prefetching.

3.3 Solutions

To address these challenges and to cope with their consequences, external schedulers can employ several solutions.

Automatic disk characterization. An external scheduler must have a detailed understanding of the specific disk for which it is scheduling requests. The only practical option is to have algorithms for automatically discovering the necessary configuration information, including LBN-to-physical mappings, seek timings, rotation speed, and command processing overheads. Fortunately, mechanisms [30] and tools [23] have been developed for exactly this purpose.

Seek conservatism. To address seek time variance and other causes of prediction errors, an external scheduler can add a small "fudge factor" to its seek time estimates. By conservatively over-estimating seek times, the external scheduler can avoid the full rotation penalty associated with under-estimation. To maximize efficiency, the fudge factor must balance the benefit of avoiding full rotations with the lost opportunities inherent to over-estimation. For freeblock scheduling decisions, a more conservative (i.e., higher) fudge factor should be selected to prefer less-utilized free bandwidth opportunities to extra full rotations suffered by foreground requests.

Resync after each request. The continuous rotation of disk platters helps to minimize the propagation of prediction errors. Specifically, when an unexpected cache hit or internal disk activity causes the external scheduler to make a misinformed decision, only one request is affected. The subsequent request's positioning delays will begin at the same rotational offset (i.e., the previous request's last sector), independent of how many unexpected rotations that the previous request incurred.

Limited command queueing. Properly utilized, command queueing at the disk can be used to increase the accuracy of external scheduler predictions. Keeping two requests at the disk, instead of just one, avoids idling of the disk head. Specifically, while one request is transferring data over the bus, the other can be using the disk head.

In addition to improving efficiency, the overlapping of bus transfer with mechanical positioning simplifies the task of the external scheduler, allowing it to focus on

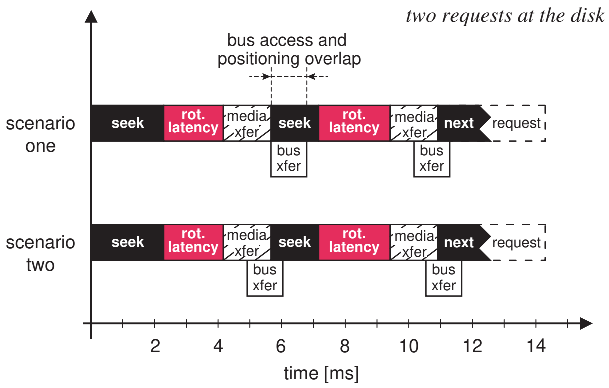


Figure 4: **Limited command queueing.** This figure repeats the two scenarios from Figure 2 but with two requests outstanding at the drive. That is, the scheduler keeps two requests at the disk — in this example, request *A* is being serviced while request *B* is queued. The drive completely overlaps the bus transfer of request *A* with the seek of request *B*, eliminating head idle time. Also, notice that the rotational latency is the same in both scenarios, making predictions easier for foreground and freeblock schedulers.

media access delays as though the bus and processing overheads were not present. When the media access delays dominate, these other overheads will always be overlapped with another request's media access (see Figure 4).

The danger with using command queueing is that the firmware's scheduling decisions may override those of the external scheduler. This danger can be avoided by allowing only two requests outstanding at a time, one in service and one in the queue to be serviced next.

Request merging. When scheduling a freeblock access to the same track as a foreground request, the two requests should be merged if possible (i.e., they are sequential and are of the same type). Not only will this merging avoid the misinformed prefetch consequence discussed above, but it will also reduce command processing overheads.

Appending a freeblock access to the end of the previous foreground request can hurt the foreground request since completion will not be reported until both requests are done. This performance penalty is avoided if the freeblock access is prepended to the beginning of the next foreground request.

4 Implementation

This section describes our implementation of an external freeblock scheduler and its integration into the FreeBSD 4.0 kernel.

4.1 Architecture

Figure 5 illustrates our freeblock scheduler's architecture, which consists of three major parts: a foreground scheduler, a freeblock scheduler, and a common dispatch queue that holds requests selected by the two schedulers.

The foreground scheduler keeps up to two requests in the dispatch queue; the remaining pending foreground requests are kept in a pool. When a foreground request completes, it is removed from the dispatch queue, and a new request is selected from the pool according to the foreground scheduling policy. This newly-selected request is put at the end of the dispatch queue. Such just-in-time scheduling allows the scheduler to consider recent requests when making decisions.

The freeblock scheduler keeps a separate pool of pending freeblock requests. When invoked, it inspects the dispatch queue and, if there is a foreground request waiting to be issued to the disk, it identifies a suitable freeblock candidate from its pool. The identified freeblock request is inserted ahead of the foreground request. The freeblock scheduler will continue to refine its choice in the background, if there is available CPU time. The device driver may send the current best freeblock request to the disk at any time. When it does so, it sets a flag to tell the freeblock scheduler to end its search.

Whenever there are fewer than two requests at the disk, the device driver issues the next request in the dispatch queue. By keeping two requests at the disk, the driver achieves the desired overlapping of bus and media activities. By keeping no more than two, it avoids reordering within the disk firmware; at any time, one request may be in service and the other waiting at the disk.

The diagram in Figure 5 shows a situation when there are two outstanding requests at the disk: a freeblock request `fb1` is currently being serviced and a foreground request `fore1` is queued at the disk. When the disk completes the freeblock request `fb1`, it immediately starts to work on the already queued request `fore1`. When the device driver receives the completion message for `fb1`, it issues the next request, labeled `fb2`, to the disk. It also sets the "stop" flag to inform the freeblock scheduler. When the foreground request `fore1` completes, the device driver sends `fore2` to the disk, tells the foreground scheduler to select a new foreground request, and (if appropriate) invokes the freeblock scheduler.

4.2 Foreground scheduler

Our foreground scheduler implements three scheduling algorithms: SSTF, SPTF, and SPTF-SW$n\%$. SSTF is representative of the seek-reducing algorithms used by many external schedulers. SPTF yields lower foreground service times and lower rotational latencies than SSTF;

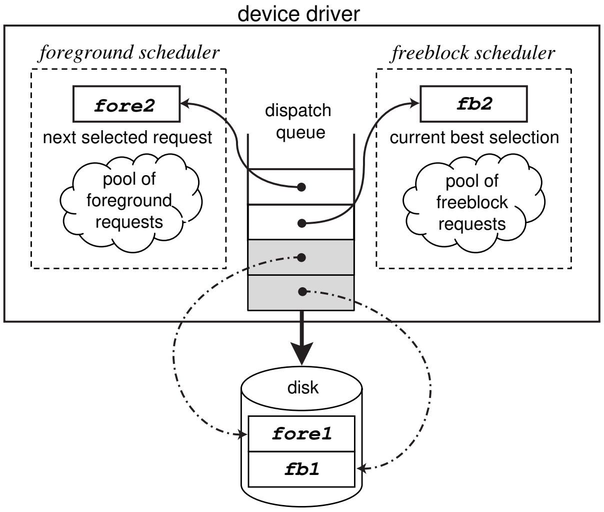


Figure 5: **Freeblock scheduling inside a device driver.**

SPTF requires the same detailed disk knowledge needed for freeblock scheduling. SPTF-SW$n\%$ was proposed to select requests with both small total positioning delays and large rotational latency components [14]. It selects the request with the smallest seek time component among the pending requests whose positioning times are within $n\%$ of the shortest positioning time.

Request timing predictions. For the SPTF and SPTF-SW$n\%$ algorithms, the foreground scheduler predicts request timings given the current head position. Specifically, it predicts the amount of time that the disk head will be dedicated to the given request; we call this time *head time*. When using command queueing, the bus activity is overlapped with positioning and media access, reducing the head time to seek time, rotational latency, and media transfer. Figure 6 illustrates the head time components that must be accurately predicted by the disk model.

The disk model in our implementation is completely parametrized; that is, there is no hard-coded information specific to a particular disk drive. The parameters fall into three categories: complete layout information with slipping and defects, seek profile, and head switch time. All of these parameters are extracted automatically from the disk using the DIXtrac tool [23]. The seek profile is used for predicting seek times, and the layout information and head switch time are used for predicting rotational latencies and media transfer times.

The layout information is a compact representation of all LBN mappings to the physical sector locations (described by a sector-head-cylinder tuple). It includes information about defects and their handling via slipping or remapping to spare sectors. It also includes skews between two successive LBNs mapped across a track,

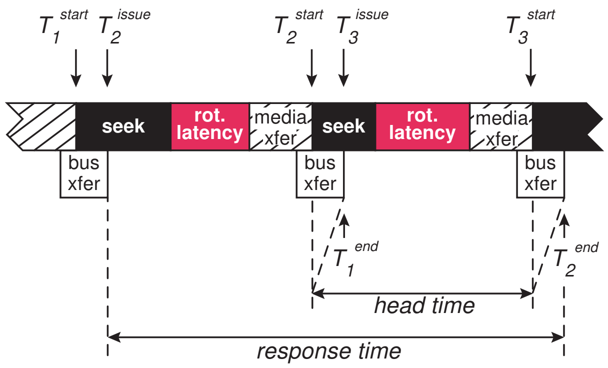


Figure 6: **Computing head time.** The head time is $T_2^{end} - T_1^{end}$. T^{issue} is the time when the request is issued to the disk, T^{start} is when the disk starts servicing the request, and T^{end} is when completion is reported. Notice that T^{issue} is different from T^{start} and that total response time, $T_2^{end} - T_2^{issue}$ includes (a portion) of bus transfer and the time the request is queued at the disk.

cylinder, or zone boundary. To achieve the desired prediction accuracy, the skews are recorded as a fraction of a revolution—using just an integral number of sectors does not give the required resolution.

The seek profile is a lookup table that gives the expected seek time for a given distance in cylinders. The table includes more values for shorter seek distances (every distance between cylinder 1–10, cylinders, every 2^{nd} for 10–20, every 5^{th} for 20–50, every 10^{th} for 50–100, every 25^{th} for 100–500, and every 100^{th} for distance beyond 500). Values not explicitly listed in the table are interpolated. Since the listed seek times are averages of seeks of a given distance, a specific seek time may differ by tens of μs depending on the distance and the conditions of the drive. Thus, the scheduler may include an explicit conservatism value to account for this variability.

4.3 Freeblock scheduler

The freeblock scheduler computes the rotational latency for the next foreground request, and determines which pending freeblock request could be handled in this opportunity. Determining the latter involves computing the extra seek time involved in going to the candidate's location and determining whether all of the necessary blocks could be fetched in time to seek to the location of the foreground request without causing a rotational miss.

The current implementation of our freeblock scheduling algorithm focuses on the goal of scanning the entire disk by touching each block of the disk exactly once. Therefore, it keeps a bitmap of all blocks with the already-touched blocks marked. When a suitable set of blocks is selected from the bitmap, the freeblock scheduler creates a disk request to read them.

The scheduling algorithm greedily tries to maximize the number of blocks read in each opportunity. To reduce search time, it searches the bitmap, looking for the most promising candidates. It starts by considering the source and destination tracks (the locations of the current and next foreground requests) and then proceeds to scan the tracks closest to the two tracks. It keeps scanning progressively farther and farther away from the source and destination tracks until it is notified via the stop flag or reaches the end of the disk. If a better free bandwidth opportunity is found, the scheduler creates a new request that replaces the previous best selection.

In early experimentation, we found that two requests on the same track often trigger aggressive disk prefetching. When the foreground workload involves sequentiality, this can be highly beneficial. Unfortunately, a freeblock request to the same track can make a random foreground workload appear to have some locality. In such cases, the disk firmware may incorrectly assume that aggressive prefetching would improve performance.

To avoid such incorrect assumptions, our freeblock scheduling algorithm will not issue a separate request on the same track. To reclaim some of the flexibility lost to this rule, it will coalesce same-track freeblock fetches with the next foreground request. That is, it will lower the starting LBN and increase the request size when blocks on the destination track represent the best selection. When the merged request completes, the data are split appropriately.

Request merging only works when the selected freeblock request is on the same (destination) track as the next foreground request. Recall that the in-service foreground request cannot be modified, since it is already queued at the disk. For this reason, our freeblock scheduler will not consider a request that would be on the source track.

Avoiding incorrect triggering of the prefetcher also prevents another same-track case: any freeblock opportunity that spans contiguous physical sectors that hold non-contiguous ranges of LBNs (i.e., they cross the logical beginning of the track). To read all of the sectors would require two distinct requests, because of the LBN-based interface. However, since these two freeblock requests might trigger the prefetcher, the algorithm considers only the larger of the two.

4.4 Kernel implementation

We have integrated our scheduler into the FreeBSD 4.0 kernel. For SCSI disks (`/dev/da`), the foreground scheduler replaces the default C-LOOK scheduler implemented by the `bufqdisksort()` function. Just like the default C-LOOK scheduler, our foreground scheduler is called from the `dastart()` function and it puts requests onto the device's queue, *buf_queue*, which is the dispatch queue in Figure 5. This queue is emptied by `xpt_schedule()`, which is called from `dastart()` immediately after the call to the scheduler.

The only architectural modification to the direct access device driver is in the return path of a request. Normally, when a request finishes at the disk, the `dadone()` function is called. We have inserted into this function a callback to the foreground scheduler. If the foreground scheduler selects another request, it calls `xpt_schedule()` to keep two requests at the disk. When the callback completes, `dadone()` proceeds normally.

The freeblock scheduler is implemented as a kernel thread and it communicates with the foreground scheduler via a few shared variables. These variables include the *restart* and *stop* flags and the pointer to the next foreground request for which a freeblock request should be selected.

Before using the freeblock scheduler on a new disk, the disk performance attributes for the disk model must first be obtained by the DIXtrac tool [23]. This one time cost of 3–5 minutes can be a part of an augmented *newfs* process that stores the attributes along with the superblock and inode information.

The current implementation generates freeblock requests for a disk scan application from within the kernel. The full disk scan starts when the disk is first mounted. The data received from the freeblock requests do not propagate to the user level.

4.5 User-level implementation

The scheduler can also run as a user-level application. In fact, the FreeBSD kernel implementation was originally developed as a user-level application under Linux 2.4. The user-level implementation bypasses the buffer cache, the file system, and the device driver by assembling SCSI commands and passing them directly to the disk via Linux's SCSI generic interface.

In addition to easier development, the user-level implementation also offers greater flexibility and control over the location, size, and issue time of foreground requests during experiments. For the in-kernel implementation, the locations and sizes of foreground accesses are dictated by the file system block size and read-ahead algorithms. Furthermore, the file system cache satisfies many requests with no disk I/O. To eliminate such variables from the evaluation of the scheduler effectiveness, we use the user-level setup for most of our experiments.

	Quantum Atlas 10k	Seagate Cheetah 18LP
Year	1999	1998
RPM	10000	10016
Head switch (ms)	0.8	1.0
Avg. seek (ms)	5.0	5.4
Number of heads	6	6
Sectors per track	334–224	360–230
Bandwidth (MB/s)	27–18	28–18
Capacity (GB)	9	9
Zero-latency access	yes	no

Table 1: **Disk characteristics.**

5 Evaluation

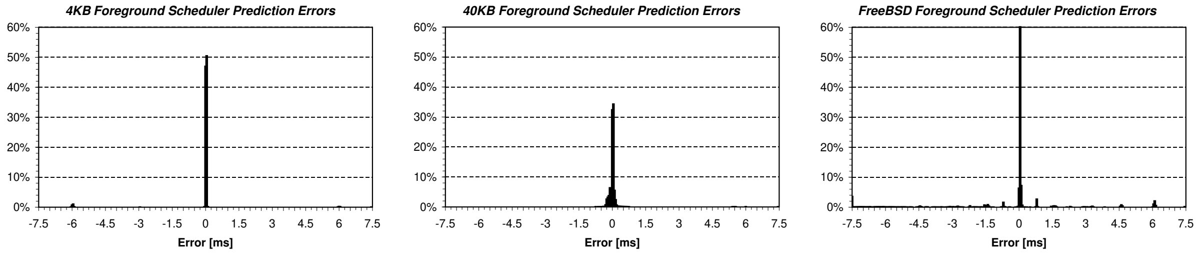
This section evaluates the external freeblock scheduler, showing that its service time predictions are very accurate and that it is therefore able to extract substantial free bandwidth. As expected, it does not achieve the full performance that we believe could be achieved from within disk firmware — it achieves approximately 65% of the predicted free bandwidth. The limitations are explained and quantified.

5.1 Experimental setup

Except where otherwise specified, our experiments are run on the Linux version of the scheduler. The system hardware includes a 550MHz Pentium III, 128 MB of main memory, an Intel 440BX chipset with a 33MHz, 32bit PCI bus, and an Adaptec AHA-2940 Ultra2Wide SCSI controller. The experiments use 9GB Quantum Atlas 10k and Seagate Cheetah 18LP disk drives, whose characteristics are listed in Table 1. The system is running Linux 2.4.2. The experiments with the FreeBSD kernel implementation use the same hardware.

Unless otherwise specified, the experiments use a synthetic foreground workload that approximates observed OLTP workload characteristics. This synthetic workload models a closed system with per-task disk requests separated by think times of 30 milliseconds. The experiments use a multiprogramming level of ten, meaning that there are ten requests active in the system at any given point. 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 KB chosen from an exponential distribution with a mean of 8 KB. Validation experiments (in [21]) 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 workload consists of a single freeblock



(a) (b) (c)

Figure 7: **PDFs of prediction error for foreground requests on a Quantum Atlas 10k disk.** The three graphs show the distribution of differences between the scheduler's predicted head time and the observed time. Negative values denote over-estimation, which means that the scheduler predicted a longer service time than was measured. The first graph shows the distribution of prediction errors for the user-level foreground workload with 4KB average request size. The second graph shows the distribution of prediction errors for the user-level foreground workload with 40KB average request size. The third graph shows the distribution of prediction errors for the FreeBSD system running the random small file read workload.

read request for the entire capacity of the disk. That is, the freeblock scheduler is asked to fetch each disk sector once, but with no particular order specified.

5.2 Service time prediction accuracy

Central to all fine-grain scheduling algorithms is the ability to accurately predict service times. Figure 7 shows PDFs of error in the external scheduler's head time predictions for the Atlas 10k disk. For random 4 KB requests, 97.5% of requests complete within 50 μs of the scheduler's prediction. The other 1.8% of requests take one rotation longer than predicted, because the seek time was slightly underpredicted and the remaining 0.7% took one rotation shorter than predicted. For the Cheetah 18LP disk, 99.3% of requests complete within 50 μs of the scheduler's prediction and the other 0.7% take one rotation longer or shorter than predicted. We have verified that more localized requests (e.g., random requests within a 50 cylinder range) are predicted equally well.

For random 40 KB requests to the Atlas 10k disk, 75% of requests complete within 150 μs of the scheduler's predictions. The disk head times for larger requests are predicted less accurately mainly because of variation in the overlap of media transfer and bus transfer. For example, one request may overlap by 100 μs more than expected, which will cause the request completion to occur 100 μs earlier than expected. In turn, because the next request's head time is computed relative to the previous request's end time, this extra overlap will usually cause the next request prediction to be 100 μs too low. (Recall that media transfers always end at the same rotational offset, normalizing such errors.) But, because the prediction errors are due to variance in bus-related delays rather than media access delays, they do not effect the external scheduler's effectiveness; this fact is particularly important for freeblock scheduling, which explicitly tries to create large background transfers.

The FreeBSD graph in Figure 7(c) shows the prediction error distribution for a workload of 10,000 reads of randomly chosen 3 KB files. For this workload, the file system was formatted with a 4 KB block size and populated with 2000 directories each holding 50 files. Even though a file is chosen randomly, the file system access pattern is not purely random. Because of FFS's access to metadata that is in the same cylinder group as the file, some accesses are physically localized or even to the same track, which can trigger disk prefetching.

For this workload, 76% of all requests were correctly predicted within 150 μs. 5% of requests, at $\pm$800 μs, are due to bus and media overlap mispredictions. There are 4% of +6 ms mispredictions that account for an extra full rotation. An additional 4% of requests at -7.5 ms misprediction were disk cache hits. Finally, 8% of the requests are centered around $\pm$1.5 and $\pm$4.5 ms. These requests immediately follow surprise cache hits or unexpected extra rotations and are therefore mispredicted.

To objectively validate the external scheduler, Figure 8 compares the three external algorithms (SSTF, SPTF, and SPTF-SW60%) with the disk's in-firmware scheduler. As expected, SPTF outperforms SPTF-SW60% which outperforms SSTF, and the differences increase with larger queue depths. The external scheduler's SPTF matches the Atlas 10k's ORCA scheduler [20] (apparently an SPTF algorithm), indicating that their decisions are consistent. We observed the same consistency between the external scheduler's SPTF and the Cheetah 18LP's in-firmware scheduler.

5.3 Freeblock scheduling effectiveness

To evaluate the effectiveness of our external freeblock scheduler, we measure both foreground performance and achieved free bandwidth. We hope to see significant free bandwidth achieved and no effect on foreground performance.

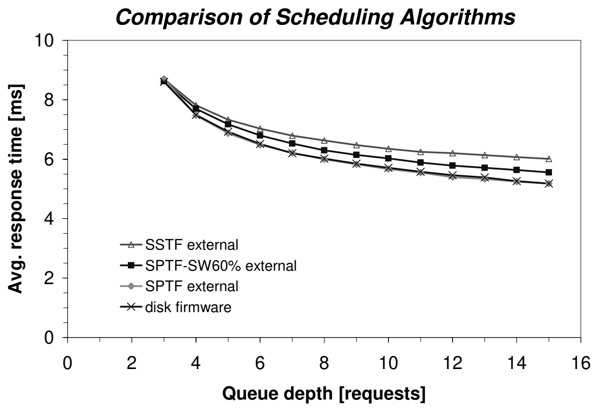


Figure 8: **Measured performance of foreground scheduling algorithms on a Quantum Atlas 10k disk.** The top three lines represent the external scheduler using SSTF, SPTF-SW60% and SPTF. The fourth line shows performance when all requests are given immediately to the Quantum Atlas 10k, which uses its internal scheduling algorithm. The "disk firmware" line exactly overlaps the "SPTF external" line, simultaneously indicating that the firmware uses SPTF and that the external scheduler makes good decisions. Linux's default limit on requests queued at the disk is 15 (plus one in service).

How well it works. Figure 9 shows both performance metrics as a function of the freeblock scheduler's seek conservatism. This conservatism value is only added to the freeblock scheduler's seek time predictions, reducing the probability that it will under-estimate a seek time and cause a full rotation. As conservatism increases, foreground performance approaches its no-freeblock-scheduling value. Foreground performance is reduced by $<2\%$ at 0.3 ms of conservatism and by $<0.6\%$ at 0.4 ms. The corresponding penalties to achieved free bandwidth are 3% and 10%.

All three foreground scheduling algorithms are shown in Figure 9. As expected, the highest foreground performance and the lowest free bandwidth are achieved with SPTF. SSTF's foreground performance is 13–15% lower, but it provides for 2.1–2.6$\times$ more free bandwidth. SPTF-SW60% achieves over 80% of SSTF's free bandwidth with only a 5–6% penalty in foreground performance relative to SPTF, offering a nice option if one is willing to give up small amounts of foreground performance.

Limitations of external scheduling. Having confirmed that external freeblock scheduling is possible, we now address the question of how much of the potential is lost. Figure 10 compares the free bandwidth achieved by our external scheduler with the corresponding simulation results [14], which remain our optimistic expectation for in-firmware freeblock scheduling. The results show that there is a substantial penalty ($\approx$35%) for external scheduling.

The penalty comes from two sources, with each responsible for about half. The first source is conservatism; its direct effect can be seen in the steady decline of the simulation line. The second source is our external scheduler's inability to safely issue distinct commands to the same track. When we allow it to do so, we observe unexpected extra rotations caused by firmware prefetch algorithms that are activated. We have verified that, beyond conservatism of 0.3 ms, the vertical difference between the two lines is almost entirely the result of this limitation; with the same one-request-per-track limitation, the simulation line is within 2–3% of the measured free bandwidth beyond 0.3 ms of conservatism.

Disallowing distinct freeblock requests on the source or destination tracks creates two limitations. First, it prevents the scheduler from using free bandwidth on the source track, since the previous foreground request is always previously sent to the disk and cannot subsequently be modified. (Recall that request merging allows free bandwidth to be used on the destination track without confusing the disk prefetch algorithms.) Second, and more problematic, it prevents the scheduler from using free bandwidth for blocks on both sides of a track's end. Figure 11 shows a free bandwidth opportunity than spans LBNs 1326–1334 at the end of a track and LBNs 1112–1145 at the beginning of the same track. To pickup the entire range, the scheduler would need to send one request for 9 sectors starting at LBN 1326 and a second request for 34 sectors at LBN 1112. The one-request restriction allows only one of the two. In this example, the smaller range is left unused.

5.4 CPU overhead

To quantify the CPU overhead of freeblock scheduling, we measured the CPU load on FreeBSD for the random small file read workload under three conditions. First, we established a base-line for CPU utilization by running unmodified FreeBSD with its default C-LOOK scheduler. Second, we measured the CPU utilization when running our foreground scheduler only. Third, we measured the CPU utilization when running both the foreground and freeblock schedulers.

The CPU utilization for unmodified FreeBSD was 5.1% and 5.4% for our foreground scheduler. Therefore, with negligible CPU overhead (of 0.3%), we are able to run an SPTF scheduler. The average utilization of the system running both the foreground and the freeblock schedulers was 14.1%. Subtracting the base line CPU utilization of 5.1% when running the workload gives 9% overhead for freeblock scheduling. In future work, we expect algorithm refinements to reduce this CPU overhead substantially.

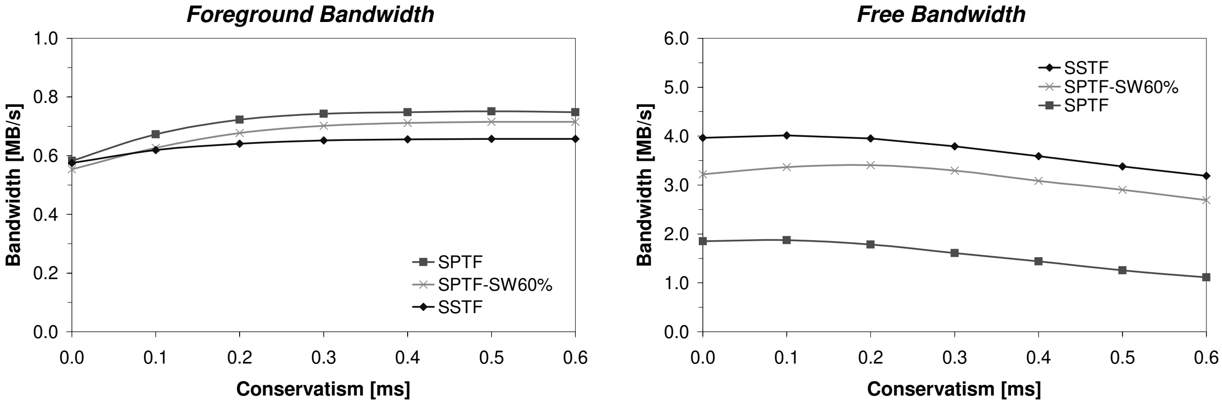


Figure 9: **Foreground and free bandwidth for a Quantum Atlas 10k as a function of seek conservatism.** The conservatism is only for freeblock scheduling decisions, which must strive to avoid overly-aggressive predictions that penalize the foreground workload. At 0.3 ms, foreground performance is 1–2% lower. At 0.4 ms, foreground performance is 0.2–0.6% lower. Note that ensuring minimal foreground impact does come at a cost in achieved free bandwidth.

Comparing the foreground and free bandwidths for the SPTF-SW60% scheduler in Figure 9 for a conservatism of 0.4 ms, the modest cost of 8% of the CPU is justified by a 6× increase in disk bandwidth utilization.

6 Related Work

Before the standardization of abstract disk interfaces, like SCSI and IDE, fine-grained request scheduling was done outside of disk drives. Since then, most external schedulers have used less-detailed seek-reducing algorithms, such as C-LOOK and Shortest-Seek-First. Even these are only approximated by treating LBNs as cylinder numbers [30].

Several research groups [1, 3, 5, 6, 26, 28, 31] have developed software-only external schedulers that support fine-grained algorithms, such as Shortest-Positioning-Time-First. Our foreground scheduler borrows its structure, its rotational position detection approach, and its use of conservatism from these previous systems. Our original pessimism regarding the feasibility of freeblock scheduling outside the disk also came from these projects—their reported experiences suggested conservatism values that were too large to allow effective freeblock scheduling. Also, some only functioned well on old disks, for large requests, or with the on-disk cache disabled. We have found that effective external freeblock scheduling requires the additional refinements described in Section 3, particularly the careful use of command queueing and the merging of same-track requests.

This paper and its related work section focus mainly on the challenge of implementing freeblock scheduling outside the disk. Lumb et al. [14] discuss work related to freeblock scheduling itself.

7 Summary

Refuting our original pessimism, this paper demonstrates that it is possible to build an external freeblock scheduler. From outside the disk, our scheduler can replace many rotational latency delays with useful background media transfers; further, it does this with almost no increase (less than 2%) in foreground service times. Achieving this goal required greater accuracy than could be achieved with previous external SPTF schedulers, which our scheduler achieves by exploiting the disk's command queueing features. For background disk scans, over 3.1 MB/s of free bandwidth (15% of the disk's total media bandwidth) is delivered, which is 65% of the simulation predictions from previous work.

Given previous pessimism that external freeblock scheduling was not possible, achieving 65% of the potential is a major step. However, our results also indicate that there is still value in exploring in-firmware freeblock scheduling.

Acknowledgements

We thank Peter Honeyman (our shepherd), John Wilkes, the other members of the Parallel Data Lab, and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support. This work is partially funded by an IBM Faculty Partnership Award and by the National Science Foundation.

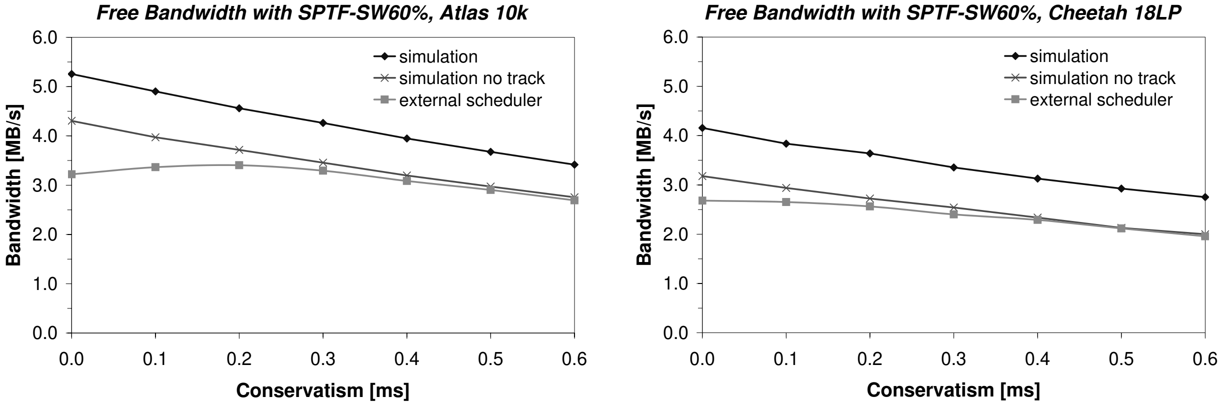


Figure 10: **Measured and simulated free bandwidth as a function of conservatism.** The line labeled simulation shows the expected free bandwidth obtained from our simulated, in-firmware freeblock scheduler operating at the given level of conservatism. The line labeled simulation no track shows a case when the simulated freeblock scheduler does not put a non-merged freeblock request on the same track as a foreground request, mimicking a major limitation of our external scheduler. The line labeled external scheduler shows the actual measured free bandwidth obtained from a disk by our freeblock scheduler implementation.

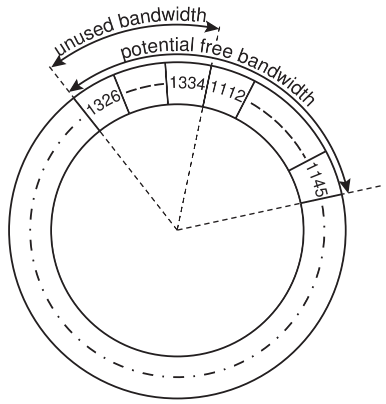


Figure 11: **A limitation of the external scheduler.** This diagram illustrates a case where the potential free bandwidth spans the start/end of a track. In this case, no single contiguous LBN range covers the potential free bandwidth. Two requests would be needed, one to LBN 1326 and one to LBN 1112. Since our scheduler can only send one free bandwidth request per track, the system will select the range from LBNs 1112-1145. This wastes the opportunity to access LBNs 1326-1334.

References

[1] M. Aboutabl, A. Agrawala, and J.-D. Decotignie. Temporally determinate disk access: an experimental approach. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI, 22–26 June 1998). Published as *Performance Evaluation Review*, **26**(1):280–281. ACM, 1998.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. 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, 1992.

[3] P. Barham. A fresh approach to file system quality of service. *International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, MO, 19–21 May 1997), pages 113–122. IEEE, 1997.

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

[5] P. Bosch and S. J. Mullender. Real-time disk scheduling in a mixed-media file system. *Real-Time Technology and Applications Symposium* (Washington D.C., USA, 31 May – 02 June 2000), pages 23–32. IEEE, 2000.

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. *IEEE International Conference on Multimedia Computing and Systems* (Florence, Italy, 07–11 June 1999), pages 400–405. IEEE, 1999.

[7] 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.

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

[9] P. 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.

[10] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 201–212. USENIX Association, 1995.

[11] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. *Summer USENIX*

Technical Conference (Boston, MA, June 1994), pages 197–207. USENIX Association, 1994.

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

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

[14] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[15] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. 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.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[17] A. G. Merten. *Some quantitative techniques for file organization*. PhD thesis. University of Wisconsin, Computing Centre, June 1970.

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

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[20] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB SCSI product manual*, Document number 81-119313-05, August 1999.

[21] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD International Conference on Management of Data* (Dallas, TX, 14–19 May 2000), pages 13–21, 2000.

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

[23] J. Schindler and G. R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[24] 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.

[25] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.

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

[27] L. Shriver. *A formalization of the attribute mapping problem*. Technical Report HPL–1999–127. Hewlett–Packard Laboratories, 1999.

[28] Trail. http://www.ecsl.cs.sunysb.edu/trail.html.

[29] R. Y. Wang, T. E. Anderson, and M. D. Dahlin. *Experience with a distributed file system implementation*. Technical Report CSD–98–986. University of California at Berkeley, January 1998.

[30] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), 1994.

[31] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.