Design and implementation of a freeblock subsystem

Eno Thereska, Jiri Schindler, Christopher R. Lumb, John Bucy,
Brandon Salmon, Gregory R. Ganger

CMU-PDL-03-107
December 2003

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Freeblock scheduling allows background applications to access the disk without affecting primary system activities. This paper describes a complete freeblock subsystem, implemented in FreeBSD. It details new space- and time-efficient algorithms that make freeblock scheduling useful in practice. It also describes algorithm extensions for using idle time, dealing with multi-zone disks, reducing fragmentation, and avoiding starvation of the inner- and outer-most tracks. The result is an infrastructure that efficiently provides steady disk access rates to background applications, across a range of foreground usage patterns.

Acknowledgements: We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support.

Keywords: scheduling, disks, asynchronous, interfaces

1 Introduction

Many background disk maintenance applications (e.g., back-up, defragmentation, and report generation) are highly disruptive if not run during off-peak hours. Others (e.g., flushes in write-back caches and failed disk reconstruction in disk arrays) rate control their activity in an attempt to bound their impact on foreground work. Various approaches exist for such rate control, most of them ad hoc, and little system support exists for removing the implementation burden from the application writer.

A recent paper [25] describes application programming interfaces (APIs) that allow disk-intensive background activities to be built cleanly and easily. The APIs allow applications to describe sets of disk locations that they want to read or write, referred to as “freeblock tasks”, and a “freeblock subsystem” will opportunistically access the disk during idle periods and rotational latency gaps. The APIs are explicitly asynchronous and they utilize dynamic memory management, just-in-time locking, and explicit rate control to encourage implementors to expose as much work as possible. That paper described how three applications (backup, write-back cache destaging, and disk layout reorganization) were converted to the APIs, and it evaluated the resulting benefits. All were shown to execute in busy systems with minimal impact ($< 2\%$) on foreground disk performance. In fact, by modifying FreeBSD’s cache to write dirty blocks for free, the average read cache miss response time was decreased by 15–30%.

This paper details the freeblock subsystem used in that work. It complements [25] and prior work [15, 16] by describing and evaluating the new scheduling algorithms and policies that make it a viable OS component[1]. Implemented in FreeBSD, our freeblock subsystem replaces the disk scheduler in the SCSI device driver. As illustrated in Figure 1, it includes both a traditional scheduler for normal (foreground) requests and a background scheduler for freeblock tasks. The background scheduler utilizes freeblock scheduling [16] and any disk idle time to weave freeblock accesses into the foreground request stream with minimal disruption.

Although previous researchers have demonstrated the ability to perform the necessary low-level disk scheduling outside the disk [1, 4, 15, 23, 28], engineering a practical freeblock subsystem has come with several hurdles. For example, the previous freeblock scheduling work placed no restrictions on memory or CPU usage by the scheduler, but a real system must be able to use most of its resources for actual applications. Previous freeblock scheduling work also ignored the needs of non-scanning tasks (e.g., cache write-backs), issues related to servicing multiple concurrent tasks, and the role of disk idle time. This paper describes and evaluates new algorithms that leap these hurdles.

As predicted by prior work, we find that freeblock scheduling makes it possible to provide disk access to background applications with minimal impact on foreground response times. Further, by also utilizing disk idle times, substantial fractions of the disk head’s time can be provided regardless of the workload produced by foreground activities. New “region-based” data structures and streamlined search algorithms allow the freeblock subsystem to use less than 8% of the CPU and less than 9MB of memory even for heavy burdens on large disks. They also allow the scheduler to account for the differing capacities of inner- and outer-zone tracks, reducing the tendency to leave inner-zone tracks until the very end of a scan (when they then become a bottleneck); further,

[1]A technical report version of [25] is available at http://www.pdl.cmu.edu/Freeblock/ for reviewers wishing to verify that it, and this paper, make distinct, substantial contributions.

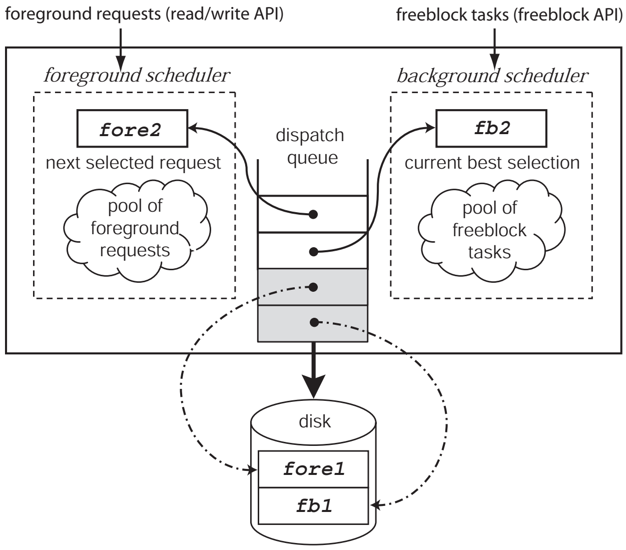


Figure 1: **The freeblock subsystem inside a device driver.**

using long idle periods to access these infrequently schedulable regions reduces overall scan times (e.g., by 20% when 30% utilized).

The remainder of this paper is organized as follows: Section 2 overviews related work. Section 3 describes the architecture of the freeblock subsystem and its integration into FreeBSD. Section 4 details and evaluates its data structures and algorithms. Section 5 summarizes the work. Section 6 discusses future work.

2 Background and related work

The freeblock subsystem described in this paper is the culmination of much prior work. This section overviews prior work and how it applies. It divides the discussion into application construction and system mechanisms that can be used for background disk-intensive applications.

2.1 Application construction

Disk maintenance activities generally have long time horizons for completion, allowing them to have lower priority at any instant than other applications running on a system. One common approach is to simply postpone such activities until expected off hours; for example, desktop backups are usually scheduled for late nights. For less sporadically-used systems, common ad hoc approaches include trickling requests a few at a time [6, 18], using idle time [10], and combinations of these with abstraction-breaking locality enhancement [11].

In [25], we described in-kernel and system-call APIs that allow disk-intensive background applications to express their needs such that a freeblock subsystem can satisfy those needs unobtrusively. The framework was shown to be effective for three very different disk maintenance activities: physical backup, cache write-backs, and layout reorganization. That paper also discusses the relationship to related storage APIs, such as dynamic sets [24], disk-directed I/O [14], and River [2]. The remainder of this subsection reviews the in-kernel API; the system-call API is similar, but without upcalls and with kernel-enforced rate control.

Applications begin an interaction with the freeblock subsystem with *fb_open*, which creates a *freeblock session*. *fb_read* and *fb_write* are used to add *freeblock tasks*, registering interest in reading or writing specific disk locations, to an open session[2]. Sessions allow applications to suspend, resume, and set priorities on collections of tasks.

No call into the freeblock scheduling subsystem waits for a disk access. Calls to register freeblock tasks return after initializing data structures, and subsequent callbacks (to the provided *callback_fn*) report task completions.

Each task has an associated *blksize*, which is the unit of data (aligned relative to the first address requested) to be returned in each *callback_fn* call. This parameter of task registration exists to ensure that reads and writes are done in units useful to the application, such as file system blocks or database pages.

Calls to register freeblock tasks do not specify memory locations. For reads, the freeblock scheduling subsystem passes back, as a parameter to *callback_fn*, pointers to buffers that it owns. When *callback_fn* returns, the buffer is reclaimed; so, if a copy is desired, the application's *callback_fn* is responsible for creating one. For writes, the associated *getbuffer_fn* is called when the freeblock scheduler selects a part of a write task. The *getbuffer_fn* either returns a pointer to the memory locations to be written or indicates that the write cannot currently be performed (e.g., because the buffer is currently locked). In this latter case, the freeblock request is not generated and that part of the freeblock task waits for a future opportunity.

The non-blocking and non-ordered nature of the interface is tailored to match freeblock scheduling's nature. Other aspects of the interface help applications increase the set of blocks asked for at once. Late-binding of memory buffers allows registration of larger freeblock tasks than memory resources would otherwise allow. For example, disk scanning tasks can simply ask for all blocks on the disk in one freeblock task. The *fb_abort* call allows task registration for more data than are absolutely required (e.g., a search that only needs one match). The *fb_promote* call allows one to convert freeblock tasks that may soon impact foreground application performance (e.g., a space compression task that has not made sufficient progress) to foreground requests. The *fb_suspend* and *fb_resume* calls allow registration of many tasks even when result processing sometimes requires flow control on their completion rate.

2.2 Mechanisms

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

[2]The term *freeblock request* is purposefully avoided in the API to avoid confusion with disk accesses scheduled inside the freeblock subsystem.

Lumb et al. [16] introduced freeblock scheduling, coined the term, and evaluated the concept via simulation. The simulations indicated that 20–50% of a never-idle disk's bandwidth could be provided to background applications with minimal effect on foreground response times. This bandwidth was shown to be more than enough for free segment cleaning in a log-structured file system or for free disk scrubbing in a transaction processing system.

Later work by two different groups [15, 28] demonstrated that outside-the-disk freeblock scheduling works, albeit with more than 35% loss in efficiency when compared to the hypothetical inside-the-disk implementation assumed in Lumb et al.'s simulations. In both cases, the freeblock scheduler was tailored to a particular application, either background disk scans [15] or writes in eager writing disk arrays [28]. In both cases, evaluation was based on I/O traces or synthetic workloads, because system integration was secondary to the main contribution: demonstrating and evaluating the scheduler. This paper builds on this prior work with numerous algorithmic enhancements needed to make freeblock scheduling effective in a real operating system.

Dimitrijević et al. [9] describe semi-preemptible I/O, which breaks up disk requests into small steps and thereby minimizes the amount of time that a higher-priority request would have to wait. Our approach to using short idle-time periods is a restricted form of this general approach.

Many disk scheduling algorithms have been devised, studied, and used. Most OSes use approximations of seek-reducing scheduling algorithms, such as C-LOOK [17] and Shortest-Seek-Time-First [8], because they are not able to predict low-level access times. Disks' firmware usually uses Shortest-Positioning-Time-First (SPTF) [12, 21]. Related to freeblock scheduling are real-time disk schedulers that use slack in deadlines to service non-real-time requests [4, 23]; the main difference is that foreground requests have no deadlines other than "ASAP," so the "slack" exists only in rotational latency gaps or idle time.

2.3 Contributions of paper

Our freeblock subsystem implementation uses outside-the-disk freeblock scheduling, any available idle time, and limited forms of semi-preemptible I/O to support the APIs described above. In addition to detailing the first complete freeblock subsystem, this paper makes several additional contributions.

It describes new time- and space-efficient algorithms and data structures that allow freeblock scheduling to work as part of a full system; previous work placed no restrictions on memory or CPU usage. In addition to efficiency improvements, it describes and evaluates algorithmic improvements for addressing zoned recording, non-scanning tasks, and fragmentation due to greediness. It describes and evaluates freeblock scheduling algorithm extensions for exploiting both short and long idle periods at the disk driver level. It describes data structures needed to support interface aspects, like suspend/resume, that have been found useful by application writers. It discusses our experiences with characterizing and modeling modern disks.

3 Architecture and integration

This section describes the architecture of our freeblock subsystem and its integration into FreeBSD.

3.1 Scheduling infrastructure

Our scheduling infrastructure replaces FreeBSD's C-LOOK scheduler. The foreground scheduler uses Shortest-Positioning-Time-First (SPTF), and the background scheduler uses rotational latency gaps and any idle time.

Like the original scheduler, our foreground scheduler is called from FreeBSD's *dastrategy()* function. When invoked, the foreground scheduler appends a request onto the driver's device queue. It then invokes the background scheduler, which may create and insert one or more freeblock requests ahead of the new foreground request.

When a disk request completes at the disk, FreeBSD's *dadone()* function is called. Into this function, we insert calls to the background and foreground schedulers. The background scheduler code determines whether the completed request satisfies any freeblock tasks and does associated processing and clean-up. The foreground scheduler selects a new foreground request, if any are pending, adds it to the dispatch queue, and invokes the background scheduler to possibly add freeblock requests. Then, *dadone()* proceeds normally.

3.2 Modeling disk behavior

Both schedulers use common library functions, which are much like other recent software-only outside-the-disk SPTF implementations [1, 3, 4, 15, 23, 28], for modeling the disk to predict positioning times for requests. Our FreeBSD implementation uses DIXtrac [20] to extract the disk parameters.

When the OS boots, the freeblock subsystem for each disk is enabled by a boot-time utility. As part of enabling the scheduler, the utility provides the parameter values needed by the scheduler's prediction models via an IOCTL() system call. These values include logical-to-physical layout descriptions, mechanical timings, and so forth. The utility gets the values from a file created by DIXtrac, which uses and builds on algorithms described by Worthington et al. [27] to obtain the needed parameter values. The program takes about 5 minutes to complete when extracting a disk with known layout algorithms (all disks, at one time, but no longer the case). For some new disks, it takes 7-8 hours to extract the layout, because DIXtrac's expert system fails and falls back on an exhaustive translation table approach. The extraction program is run once when the corresponding disk is first added to the system, before any file systems are created on it. Storing the results from one extraction is important for two reasons: adding 5 minutes to every boot time would be unacceptable, and the extraction program uses some writes meaning that data stored on the disk could be corrupted.

3.3 Background scheduler architecture

The background scheduler has a two-layer architecture, as illustrated in Figure 2. The first layer is the *task-manager layer*. Applications register tasks with this layer via the in-kernel APIs described in Section 2.1. The second layer is the *scheduling layer*. This layer is responsible for scheduling background tasks by using available disk idle time and rotational latency gaps. Because our implementation is outside-the-disk, scheduling layer must also have the disk layout and mechanical delays as described above.

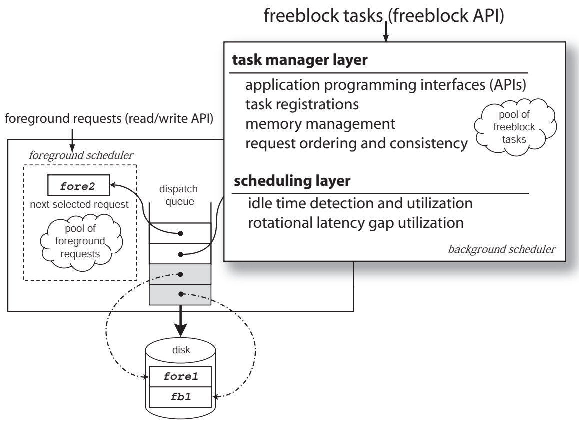


Figure 2: **Background scheduler layers.** This diagram illustrates the two layers of the freeblock subsystem's background scheduler: the task manager and scheduling layer. The task manager layer is responsible for keeping track of registered background tasks. The scheduling layer is responsible for scheduling them when appropriate.

3.4 Scheduling steps

When the freeblock scheduler decides to schedule a freeblock request for part of a registered task, it constructs a request to be placed in the disk's queue. For writes, it calls *getbuffer_fn* to get the source memory location. For reads, it allocates memory to act as the destination. When the freeblock request completes, and post-request processing (described below) is done, allocated memory is freed.

Recall that the *blksize* parameter of each registered task dictates the atomic unit of freeblock requests. Our current scheduler only generates freeblock requests at least as large as the *blksize* parameter. As a result, large values can prevent forward progress, especially if the size is larger than can be fetched outside of long idle times. Another option, for reads, would be to internally buffer portions of *blksize* units fetched with multiple freeblock requests. We have not yet encountered an example where large (>64KB) *blksize* limits are necessary, so we have not paid a price for avoiding the memory pressure and complexity associated with the buffering option. Section 4.4 analyzes in greater detail the impact of varying *blksize* on the memory complexity and scheduler efficiency.

When any disk request (foreground or freeblock) completes, control is passed to the freeblock subsystem in the device driver. The freeblock system performs a quick search to determine whether any parts of registered freeblock tasks can be satisfied. For each satisfied part of a freeblock task, the *callback_fn* is called. When the callback completes, any memory that was occupied by the satisfied part of the freeblock task is reclaimed. When the last part of a freeblock task is satisfied, associated data structures are cleaned up.

4 Freeblock subsystem algorithms

This section details the algorithms used in our freeblock subsystem. As it does so, it evaluates their effectiveness relative to previous freeblock scheduling algorithms and quantifies their CPU and memory costs. It also reports on difficulties with the "outside-the-disk" implementation beyond those anticipated in earlier works.

4.1 Experimental apparatus

The experiments throughout this paper use a workstation-class box that includes a 1GHz Pentium III, 386MB of main memory, an Intel 440BX chipset with a 33MHz, 32bit PCI bus, and an Adaptec AHA-2940 Ultra2Wide SCSI controller. Unless otherwise mentioned, the experiments use the Seagate Cheetah 36ES disk drive, whose characteristics are listed in Table 1. The system runs FreeBSD4.4[3], extended to include our freeblock subsystem.

Most of the experiments in this section are microbenchmarks, which allow more control on different aspects of the system. Unless otherwise stated, we use a synthetic workload of small 4-8KB reads and writes with a read-write ratio of 2:1, keeping the disk fully utilized. Two requests are kept outstanding at a time and a new one is generated after each completes. For exploring idle time usage and empty queue scheduling, we draw inter-arrival times from a uniform distribution with variable means.

The default background activity is that of scanning the entire disk. A single *fb_read* task is registered for all blocks on the disk, with *blksize*=32KB.

4.2 Data structures and usage

There are two sets of data structures used by the freeblock subsystem, as illustrated in Figure 3, for its two layers. The *task manager data structures* maintain task-related bookkeeping information. The *scheduling data structures* maintain direct information about the particular device that is using freeblock scheduling and about the areas of the disk that freeblock tasks span.

Task manager data structures: The first data structure in this category is the session list, which keeps information about which freeblock sessions are open. The session list also keeps, for each open session, its priority, a list of tasks in that session and their callback and getbuffer functions. The *status* field records whether the session is open or closed, suspended or schedulable.

The second data structure is a hash table that contains the registered freeblock tasks for quick lookup. Each task maintains a bitmap for blocks that still haven't been satisfied. To reduce memory constraints the freeblock subsystem has a minimum pickup size, *MIN_PICKUP_SIZE*, of blocks it tries to read from (or write to) for free. This parameter is task-specific and is a multiple of the *blksize* parameter that a task specifies when it registers. Each bit in the task bitmap represents a unit of *MIN_PICKUP_SIZE*.

In our illustration, two tasks are registered with the freeblock subsystem for block ranges 1096-1155 and 1232-1315, respectively.

Scheduling data structures: For purposes of freeblock scheduling, each cylinder of the disk is divided into angular *regions*, as illustrated in the bottom right of Figure 3. These regions are used by the freeblock algorithms to quickly determine which tasks want blocks close to the disk head location. Each disk cylinder (C cylinders in total) is divided into R angular regions for a total of RxC regions. For example, Figure 3 shows the case when R is 8, the default value used in our evaluations. Two freeblock fragments, one of them belonging to task 1 and the other to task 2, span the second, third and sixth region on a particular track.

A "task map" is kept for in every region. This map keeps track of tasks that desire blocks in that region. Whenever the disk head is in a particular region, the freeblock scheduler queries the tasks in that region to give it the range of blocks they desire to pick up, together with the *blksize*

[3]The freeblock subsystem has been integrated in FreeBSD4.4 and recently in FreeBSD5.0.

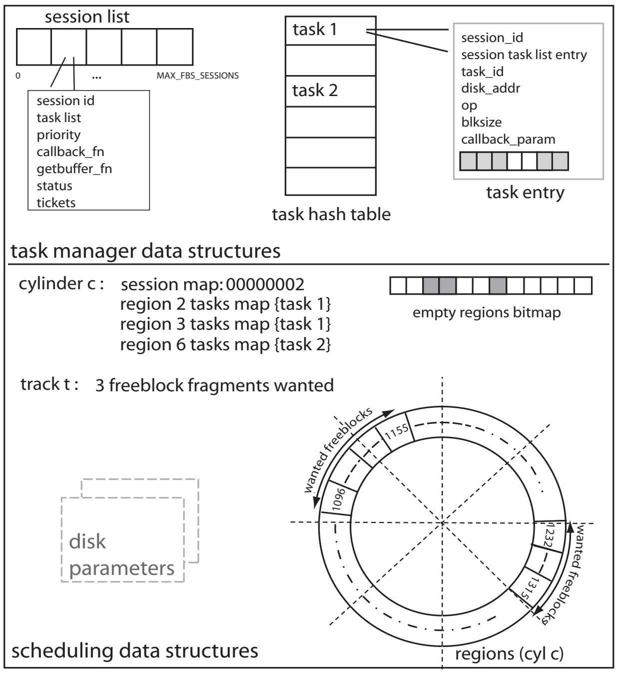


Figure 3: **Main freeblock subsystem data structures.**

parameter. The freeblock scheduler then determines how much of those blocks it can read (or write to) for free. The reason that every cylinder and not every track is divided into regions is based on the observation that if the freeblock scheduler is not able to satisfy a freeblock fragment on a particular cylinder, region and surface, it will most likely not be able to satisfy any other freeblock fragments on other surfaces (but on the same region and cylinder). This observation considerably reduces the memory cost by keeping fewer regions.

A region bitmap keeps track of regions with registered tasks and empty regions. This bitmap is used by the freeblock algorithms to quicken their search; regions with no freeblocks can be immediately pruned.

Each cylinder contains a map of sessions and the number of freeblock fragments they desire in that cylinder. There is one entry per session in the map, and the entry size is 2 bytes, allowing for up to 65536 fragments per session per cylinder. During the search for free bandwidth, cylinders with no sessions (or with suspended session) are pruned. Also, only cylinders that have sessions of a certain priority are considered, as described in Section 4.5.

Each track has a structure that records the total number of freeblock fragments on that track that are desired for free. The number of freeblock fragments is useful to skip tracks that have fewer fragments than what has already been picked up.

The disk parameter structure was discussed in Section 3.2 and is the last structure needed by the whole freeblock subsystem. If freeblock scheduling were done inside-the-disk, this structure would be replaced by the disk's existing scheduler support. In the FreeBSD implementation, this structure is about 8KB for an 18GB disk and its size scales well with the disk size.

Disk	Year	Sectors Per Track	No. of Cylinders	No. of Heads	Capacity	Tskmgr. Overhead	Scheduler Overhead
Quantum Atlas III	1997	256–168	8057	10	9 GB	**2.7MB**	**1.3MB**
IBM Ultrastar 18LZX	1999	382–195	11634	10	18 GB	**3.1MB**	**1.6MB**
Seagate Cheetah 36ES	2001	738–574	26302	2	18 GB	**3.1MB**	**3.2MB**
Maxtor Atlas 10k III	2002	686–396	31022	4	36 GB	**3.9MB**	**3.8MB**
Seagate Cheetah 73LP	2002	746–448	29550	8	73 GB	**5.7MB**	**3.9MB**

Table 1: **Freeblocks subsystem memory cost for modern disk drives.** This table shows the total memory overhead of the freeblock subsystem, when 10 large tasks, each wanting to read 100% of the disk for free, 1000 medium tasks, each wanting to read or write 0.1% of the disk for free and 10000 small tasks, each wanting to read or write 0.01% of the disk for free, are registered and schedulable. The large tasks are representative of applications such as backup or virus detection, the medium tasks are representative of applications such as disk reorganization, whereas the small tasks are representative of applications like cache write-back. In all cases, the freeblock subsystems divides each cylinder in 8 angular regions and the blksize and MIN_PICKUP_SIZE parameters for all registered tasks are 32KB.

Taskmg.	Structures	Memory (MB)
	session list	1.06
	task hash table	1.99
Scheduling.	Structures	Memory (MB)
	total # regions	2.53
	region bitmap	0.026
	cylinder session maps	0.42
	track fragments	0.10
	disk parameters	0.08

Table 2: **Detailed memory overhead for a Seagate Cheetah 36ES.** This table shows the per-data-structure memory overhead of the freeblock subsystem using the same setup at Table 1

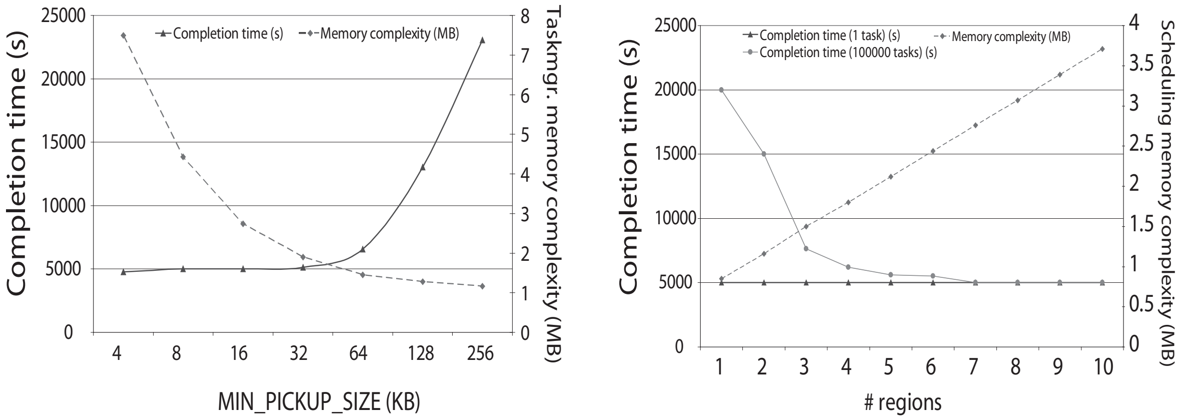
4.3 Memory and CPU overheads

Memory complexity: Table 1 shows the cost of both classes of data structures for some modern disks. Table 2 breaks down the memory cost per layer for the Seagate Cheetah 36ES disk. Both tables illustrate the scenario when the freeblock subsystem is massively loaded.

Even with such a large set of registered tasks, the memory demands are quite reasonable. The memory complexity of the task manager is mainly dependent on the *MIN_PICKUP_SIZE* parameter, which for a given task is always at least as large as (or a multiple of) the *blksize* parameter. Recall that a task maintains a bit for each unit of size *MIN_PICKUP_SIZE* and it clears that bit whenever the unit has been read (or written) for free. Increasing *MIN_PICKUP_SIZE* decreases the opportunities for finding a large enough rotational latency gap to satisfy that unit, but reduces the task manager's memory complexity. Decreasing *MIN_PICKUP_SIZE* increases the opportunities for freeblock scheduling to satisfy that unit, but places a burden on the memory complexity. Figure 4(a) shows the tradeoff involved when using the Seagate Cheetah 36ES disk.

The memory complexity of the scheduling layer is mainly dependent on the number of regions. Each region has a fixed-size map for quick lookup that contains up to N task ids[4] ; if more tasks

[4]N is 2 in our implementation



(a) **Memory complexity and scheduler's performance as a function of *MIN_PICKUP_SIZE*.**

(b) **Memory complexity and scheduler's performance as a function of the number of cylinder regions.**

Figure 4: **Memory complexity and scheduler's performance tradeoffs.**

fall into a region, they are added to a list of tasks that is otherwise uninitialized. Because there is a fixed-size map associated with each region, the greater the number of regions, the larger the memory overhead. However, decreasing the number of regions may not necessarily reduce the memory overhead. This is because the task id will have to be registered in all region(s) it falls into, independently on the number of regions there are.

Depending on the number and characteristics of the registered tasks, decreasing the number of regions may degrade the performance of the freeblock scheduler. Regions help narrow down the search for tasks that can be satisfied. Figure 4(b) shows the tradeoff involved when using the Seagate Cheetah 36ES disk. When the whole disk is to be scrubbed for free using one task, the number of regions is irrelevant. This is because for every track searched, there is only one task to be queried for wanted blocks on that track. When the complete read of the disk is split into 100000 tasks (each reading 1/100000 of the disk in consecutive order), the number of regions significantly impacts the performance of the scheduler because having fewer regions means that search time is wasted searching for tasks whose blocks cannot be read (or written to) for free and the first pass does not complete. Note that we chose 8 regions because at the host we have the available memory resources. For an inside-the-disk implementation, where memory resources are scarcer, choosing 4 regions would halve the memory requirements while losing only 15% of the scheduler's efficiency (the scheduler's efficiency inside-the-disk is, according to simulations shown in [15, 16], 35% higher then outside-the-disk, so this loss wouldn't be an issue).

CPU utilization: The CPU utilization of the freeblock scheduler varies from 0.5%, when there are no freeblock tasks registered (but the foreground scheduler is still doing SPTF), to 8% of a Pentium III 1GHz in the middle of scanning. The main source of CPU usage lies in computing physical locations and relative angular distances from logical blocks. Code analysis with the common *gprof* tool reveals that over 60% of the time used the freeblock subsystem is used by this conversion. These functions perform relatively poorly in our system because of our desire to support the wide range of differing layout schemes observed for different disks. This results in generic, thus expensive, code paths. Section 4.4 discusses efficient algorithms used to minimize the amount of CPU utilization.

4.4 Scheduling algorithms

The foreground scheduler uses the SPTF algorithm to select the next request to send to disk. The background scheduler includes algorithms for utilizing otherwise wasted rotational latency gaps and for detecting and using disk idle time.

There are many possible algorithms that could be used for utilizing rotational latency gaps. Desired characteristics for such algorithms include low CPU utilization, low memory requirements, and high utilization of free bandwidth opportunities. This section discusses several algorithms and the environments they are suited for.

To minimize the CPU utilization, all algorithms use two phases for their search. The first phase is a quick-pass search through the disk cylinders during which the algorithms find the closest freeblock fragments to the source and destination cylinders; these fragments are the easiest to pick up. The first phase runs for a *quanta* of time, whose length is derived from the desired CPU utilization. After the quick-pass search completes the freeblock subsystem yields the CPU to other processes. If there is idle CPU time the freeblock scheduler continues to refine its search for freeblock fragments. This is called the second phase of search. Depending on the number of CPU tasks in the system, the second phase may or may not happen. Hence, it is important that the first phase be very efficient and effective. The fragments are coalesced whenever possible (e.g., when they are on the same track) or sent to the disk as separate freeblock requests. In the FreeBSD implementation, the freeblock algorithms do not attempt to pick up fragments in the source track and only try to pick up fragments that can be coalesced with the destination request. The reason for this conservatism is avoiding the interference with any disk prefetching, as described in [15].

Below we discuss several freeblock algorithms and how they deal with three biases that result from the way logical blocks are physically organized on a disk:

1. *Foreground workload bias* towards high-density zones. This bias arises because the first few zones of a disk usually have a larger number of logical blocks than the middle or innermost zones. Hence, because the foreground workload tends to go to these zones, the other zones will less frequently get a chance to be searched for free bandwidth.

2. *Search bias* towards cylinders closer to the middle of the disk. With all else equal, cylinders closer to the middle of the disk have a larger chance of being searched for free bandwidth than innermost or outermost cylinders.

3. *Search bias* towards high-density cylinders. With all else equal, a bandwidth-greedy strategy for searching for free bandwidth will favor cylinders with a large concentration of logical blocks.

Whereas the first bias is workload-dependent, the other two biases can be minimized as described in the design of the algorithms below. The foreground and background workloads are the default ones.

BW-Greedy: The first algorithm, called BW-Greedy, is similar to the one described by Lumb et al. in [16]. The algorithm starts by considering the source and destination tracks (the locations of

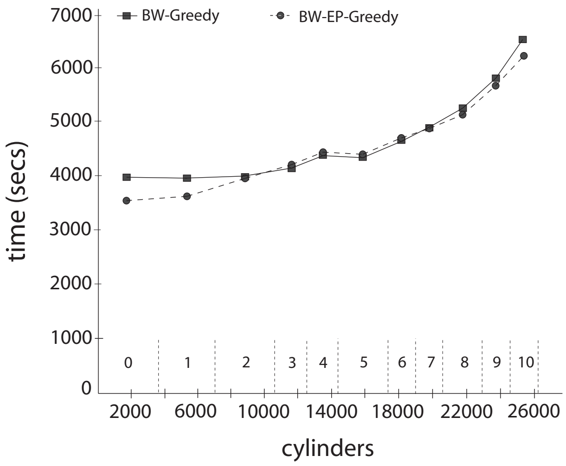


Figure 5: **A comparison of BW-Greedy and BW-EP-Greedy.** This diagram illustrates the time to pick up the last block of each of the 11 zones of the Seagate 36ES disk for both BW-Greedy and BW-EP-Greedy when the background activity is that of disk scrubbing.

the current and next foreground requests) and then proceeds to scan the tracks closest to these two tracks. The greedy approach 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?". The main advantage of the greedy approach is its relative simplicity. The main disadvantage is that it can induce severe fragmentation in the freeblock space. We have observed that, after some time, many small freeblock fragments will be needed around the disk, making searches inefficient. Another disadvantage of this algorithm is that it has a *search bias towards high-density cylinders*. This bias favors the outer cylinders over the inner cylinders. BW-Greedy also exhibits a *search bias towards cylinders closer to the middle of the disk*. This is because it searches the cylinders closest to the foreground source and destination cylinders.

BW-EP-Greedy: To alleviate the bias BW-Greedy has towards cylinders closer to the middle of the disk, BW-EP-Greedy (bandwidth-extra pass-greedy) during the quick pass partitions the search quanta such as two-thirds of it are spent searching cylinders starting from the source and destination cylinders, and the remaining time is spent searching starting from the disk's innermost and outermost cylinders. Figure 5 illustrates the difference between BW-Greedy and BW-EP-Greedy. BW-EP-Greedy still exhibits the *search bias towards high-density cylinders*.

Angle-EP-Greedy: To correct the bias BW-EP-Greedy has towards high-density cylinders, Angle-EP-Greedy compares the **angles** accessed by the potential free requests. Thus, *free angles* rather then *free blocks* are compared when selecting the best units to pick up. Figure 6 illustrates the difference between BW-EP-Greedy and Angle-EP-Greedy. The per-zone pickup time is further smoothed (but not flat, since the foreground bias towards high-density zones still exists) and results in a 15% reduction in total scan time.

Angle-EP-Frag: The fourth algorithm, called Angle-EP-Frag, is the same as Angle-EP-Greedy, except that, whenever possible, Angle-EP-Frag prefers accessing fragments that will clear a whole track. Given two equally good choices Angle-EP-Frag will prefer the one belonging to the track with the fewest free blocks remaining to be accessed. These methods reduce fragmentation considerably. Figure 7 illustrates the difference between Angle-EP-Greedy and Angle-EP-Frag.

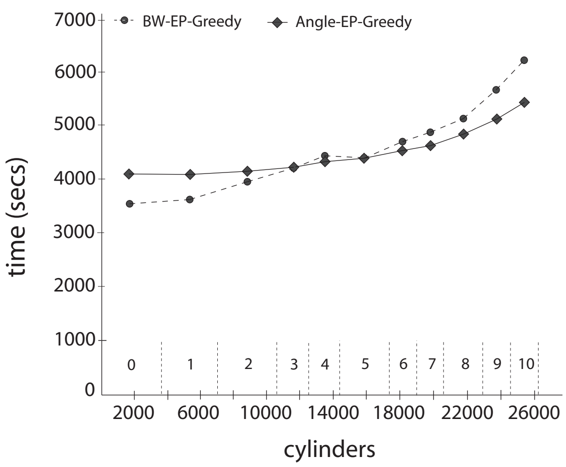


Figure 6: **A comparison of BW-EP-Greedy and Angle-EP-Greedy.** This diagram illustrates the time to pick up the last block of each of the 11 zones of the Seagate 36ES disk for both BW-EP-Greedy and Angle-EP-Greedy when the background activity is that of disk scrubbing.

Angle-EP-Frag is the default algorithm in our system. All algorithms have a low CPU utilization around 2-8%, depending on the total area of the disk wanted for free.

4.5 Priority-guided search

Angle-EP-Frag is modified to consider priorities when searching for freeblocks. The current implementation uses a simple form of lottery scheduling [26]. The initial *tickets* allocated to each session are proportional to the priorities associated to that session by the application. The lottery determines both which pending tasks are considered, since there is limited CPU time for searching, and which viable option found is selected.

The role of the tickets is two-fold. First, they provide a way to allocate the search time to tasks that belong to each priority group. Second, they act as absolute priorities when freeblock units from different category groups are picked up in one search.

During the quick-pass search Angle-EP-Frag searches cylinders with tasks from the winning session. Any option from the winning session found will be selected. In addition, all pending tasks on the destination cylinder and within one cylinder of the source are considered; these are the most likely locations of viable options, reducing the odds that the rotational latency gap goes unused. During a second phase, all pending tasks from the winning session are considered and given strict priority over pending tasks from other sessions.

4.6 Idle time integration

As the foreground workload moves from heavy to moderate or light, the opportunity to use rotational latency gaps rapidly decreases. Therefore, steady disk maintenance progress requires the ability to use idle disk time. Previous research [10, 19] suggests that most idle periods are just a few milliseconds in length, but the bulk of idle time comes in very long (multiple second) durations. This section describes how our freeblock subsystem uses short and long idle times.

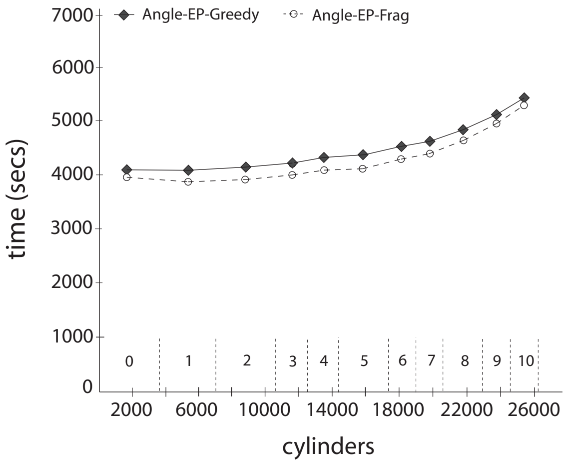


Figure 7: **A comparison of Angle-EP-Greedy and Angle-EP-Frag.** This diagram illustrates the time to pick up the last block of each of the 11 zones of the Seagate 36ES disk for both Angle-EP-Greedy and Angle-EP-Frag when the background activity is that of disk scrubbing.

In general, using idle time for background tasks raises the following concerns:

1. Interference with on-disk caching and prefetching. The interference with the on-disk cache could happen because freeblock fragments could evict other blocks in the cache that could be wanted by an upcoming foreground request. The interference with prefetching could also happen because the disk prefetching mechanism after a foreground request completes could be preempted by our freeblock scheduler issuing a freeblock request. Note that there is no interference with the file system cache since buffers returned by freeblock requests bypass that cache.

2. Delays incurred by loss of locality. The interference with locality could happen if, between two foreground requests (A and B) that exhibit high locality (meaning that the time to go from the location of A to the location of B is small), a freeblock request is inserted during idle time that is far away from both A and B.

3. Delays incurred by non-preemptibility of disk requests. Once the disk starts to service a freeblock request, it cannot be pre-emptied when a new foreground request arrives.

While the delays incurred by non-preemptibility have been recently addressed by aggressively partitioning requests as discussed in [9], the interference with the cache and locality requires more analysis, and is discussed below, together with methods to utilize short and idle time periods.

To detect idle times, the scheduler monitors the arrivals and completions of foreground requests. It maintains two separate timers: one to detect short idle time periods and one to detect long idle time periods. Figure 8 shows two example scenarios when using the idle-detector. In the first scenario, there is not enough idle time in the system to enter the long-idle state. A foreground request arrives while the system is still in the short idle state. The method we describe below minimizes the impact on subsequent foreground requests.

In the second scenario, a foreground request arrives after the system enters the long idle time state. During the time in this state, the scheduler may generate requests that are geometrically far

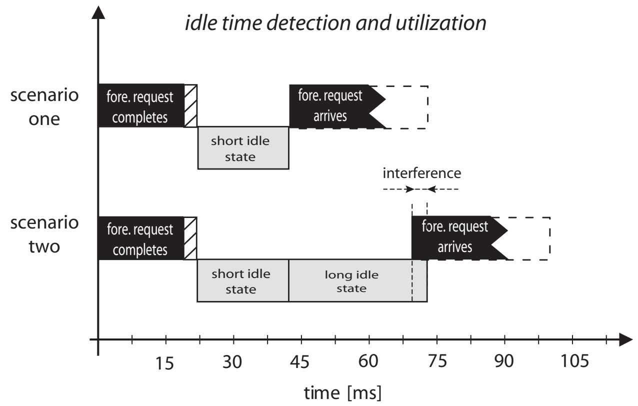


Figure 8: **Idle-time activity.** This figure shows the sequence of events that lead to idle time detection and utilization. When the last request at the disk completes, a short-idle timer is started. When the timeout of that timer expires, we have detected short idle time. In the first scenario, a foreground request arrives during the short-idle state and the idle time utilization completes. In the second scenario, after the short-idle state completes, the long-idle state begins. The next foreground request that arrives signals the end of the long-idle state.

away from the position the disk would have been to, if no freeblock tasks were satisfied. Therefore, the arriving foreground request may have to wait for the freeblock request currently at the disk to complete (concern #3) and then pay a locality penalty (concern #2). In practice, we observe minimal impact on foreground requests (less then 0.5%), given a reasonable idle detection threshold [10].[5]

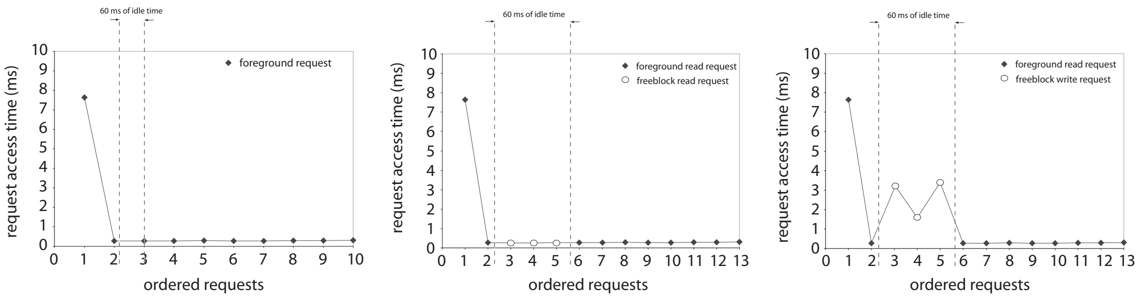
Short idle time usage: During short idle time periods, the scheduler promotes parts of freeblock tasks that fall into the same track as the one on which the disk head currently resides. Most disk drives prefetch data during idle time, and this approach allows clean interaction with such prefetching. Specifically, we can read data from the prefetch buffer after it has been fetched, without causing any mechanical delays. Further, data moved from the disk's buffers can be cached in the freeblock subsystem, avoiding concerns about displacement by use of idle time.

Figure 9 shows an example of the interaction of work done during idle time and foreground read requests. The Seagate 36ES disk performs aggressive whole-track prefetching after detecting that the first two foreground read requests are sequential. An idle time period of 60ms follows, after which 8 more foreground requests arrive wanting blocks on the same track. Because the whole track has been prefetched, the service time for these requests is very small as shown in Figure 9(a). Figure 9(b) and Figure 9(c)shows the impact of doing freeblock reads and writes on the same track during the idle time. The service time of the next 8 requests is not affected.

Long idle time usage: During long idle time periods the scheduler promotes parts of freeblock tasks that are either heavily fragmented or have been pending in the system for too long. Like previous research [15], we find the innermost and outermost cylinders are often difficult to service in rotational latency gaps. Therefore, the scheduler preferentially uses long idle time periods to service freeblock fragments that fall in those areas.

Integration with rotational latency gaps: Figure 10 shows the efficiency of the freeblock subsystem as a function of disk utilization using the default microbenchmark. The *callback_fn*

[5]The threshold we are using is 20ms



(a) **Foreground reads on a track.**

(b) **Foreground and freeblock reads on a track.**

(c) **Foreground reads and freeblock writes on a track.**

Figure 9: **Impact of freeblock activity during short idle times.** Two sequential foreground requests trigger whole-track prefetching. After 60ms of idle time, 8 more foreground requests arrive, wanting blocks on the same track. In Figure 9(a) these 8 requests are served from the cache. In Figure 9(b) three freeblock read requests are serviced on the same track, due to idle time detection. In Figure 9(c), three freeblock write requests are serviced on the same track, due to idle time detection. In all cases, the freeblock requests do not impact the service time of the subsequent 8 foreground requests.

re-registers each block as it is read. This guarantees that, at all times, there is a constant number of blocks wanted.

The freeblock subsystem ensures that background applications make forward progress, irrespective of the disk's utilization. As expected, the progress is fastest when the disk is mostly idle. The amount of free bandwidth is lowest when the system is 30-60% utilized. There are two reasons for this. First, there are few long idle periods in the system and yet there is also less rotational latency to exploit. Short idle times are less useful than these others. Second, an interesting scenario happens when the disk is under-utilized. In all previous experiments, we have assumed that there is at least one foreground request (A) being serviced at the disk. Then, another one (B) comes and a freeblock request may be squeezed in-between. The scheduling algorithms only search until the disk head arrives at A's location. In the under-utilized scenario, the disk queue is empty and a request (C) comes in. We can obviously search at most until the disk head arrives at C's location. Notice, though, that the more time we spend searching, the *less* freeblock opportunities we can consider, since the disk head is moving while we are searching. In our implementation, we only search at most for the time it takes the disk head to pass over a single region and we only consider freeblock opportunities that lay after that region. Hence, our chances of scheduling something for free are limited due to the limited search time and limited search space.

Finally, Figure 11 shows how long idle times can be used to improve the efficiency of the freeblock algorithms by more then 20%. Smart utilization of idle time significantly reduces the overall time to finish the disk scan, because during idle time the most difficult cylinders are picked up.

4.7 Scalability with disk size

Our algorithms scale well with increasing disk sizes, as shown in Figure 12. This graph shows three artificial size doublings of the original Seagate Cheetah 36ES disk (18GB): doubling linear

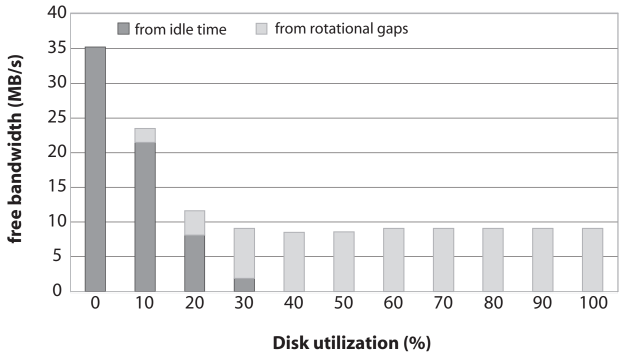


Figure 10: **Freeblock subsystem efficiency.** This diagram illustrates the instantaneous free bandwidth extracted for a background disk scan as a function of the disk's utilization. When the foreground workload is light, idle time is the main source of free bandwidth. When the foreground workload intensifies, the free bandwidth comes from rotational latency gaps.

density, doubling track density, and doubling the number of surfaces. The performance was evaluated with the DiskSim [5] simulator. It is interesting to observe that, if the increase in capacity results from an increase in linear density only (36GB, 2 heads, X cyls), then a 36GB disk can be read in approximately the same time as the original 18GB disk. If the increase comes solely as a result of an increase in cylinder density (36GB, 2 heads, 2X cyls) or increase in the number of disk platters (36GB, 4 heads, X cyls), the increase in reading the whole disk for free is only 1.5x higher then the time it takes to read the 18GB disk. This is because of the efficient region search described above, which skips searching for blocks in regions on the other heads, if wanted blocks cannot be satisfied on the current head.

4.8 Difficulties with the outside-the-disk implementation

Most difficulties that arise from the outside-the-disk implementation are previously mentioned in [15]. We briefly report the main ones here for completion, together with new issues we have encountered (and some potential solutions).

Internal disk activities: Disk firmware must sometimes execute internal functions (e.g. thermal calibration) that are independent of any external requests. These activities will occasionally invalidate the scheduler's predictions.

In-drive scheduling: An external scheduler that needs to maintain control over the order requests are executed must make sure the disk does not re-arrange the order of requests. We use limited command queuing (keeping at most 2 requests at the disk at the same time) to prevent the disk from doing any scheduling of its own.

Non-constant delays and coarse observations: An external scheduler sees only the total response time for each request. Deducing the component delays from these coarse observations is made difficult by the inherently intricate ways bus and media transfer overlaps differ. We choose to be conservative in predicting seek times to address this variance. By conservatively over-estimating seek times, the external scheduler can avoid the full rotation penalty associated with under-estimation. A "fudge factor" is added to the predicted seek time for that. This factor adaptively changes and attempts to closely track the true seek time for a request.

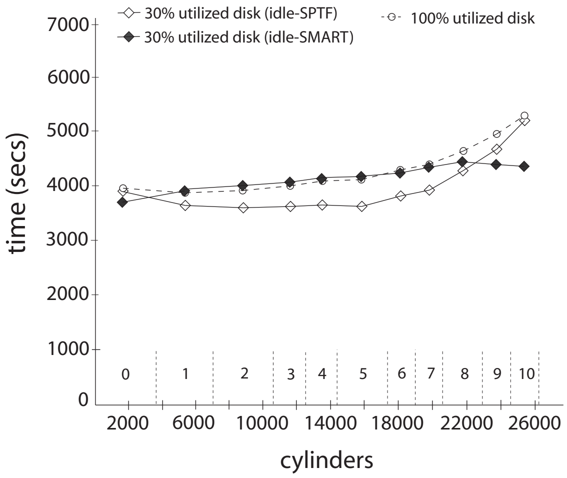


Figure 11: **Efficiently utilizing the long idle time** This diagram illustrates the time to pick up the last block of each of the 11 zones of the Seagate 36ES disk when the disk is 100% and 30% utilized. When the disk is 30% utilized there are 2 ways of using idle time. The first one (idle_SPTF) simply attempts to satisfy freeblock units close to the current disk head position. The efficient way (idle_SMART) attempts to satisfy units that fall into the innermost and outermost cylinders. By doing so, it reduces the overall running time from approximately 5000s to approximately 4000s, a 20% improvement.

	Without soft-updates				With soft-updates			
Benchmark	**ON/ON**	**ON/OFF**	**OFF/ON**	**OFF/OFF**	**ON/ON**	**ON/OFF**	**OFF/ON**	**OFF/OFF**
Postmark	1016	1021	1016	1022	886	897	884	897
TPC-C	1155	1161	1156	1142	1162	1145	1166	1144
SSH-build	63	73	63	73	49	49	49	49
CP-DIFF	34	34	34	34	36	36	36	36

Table 3: **Performance of four benchmarks as a function of the disk cache policies.** This table illustrates the total time of completion for Postmark, SSH-build and CP-DIFF and the transactions per minute for TPC-C as a function of the metadata update policy (synchronous or using soft-updates) and on-disk cache policy. CP-DIFF is a benchmark that copies two large (100MB) trace files to the partition and diffs them line by line. The disk cache policy is described by the (read cache policy / write cache policy) tuple. For the write cache policy, ON means the policy is write-back whereas OFF means the policy is write-through. The disk used is Seagate Cheetah 36ES. The disk has a 4MB on-board cache.

On-board caching and prefetching: The main difficulties with the outside-the-disk implementation of freeblock scheduling stem from on-board read and write caches present in modern disk drives. These caches store disk blocks according to often-complex vendor-specific policies.

Figure 13 illustrates what happens to the access predictions for write requests when the disk cache uses the write-back policy. The prediction accuracies deteriorate and freeblock scheduling is no longer possible. In our implementation, we disable the write-back policy of the disk. It is not in the scope of this paper to evaluate the exact impact on the performance of the system when the write-back policy is off. However, we believe the impact is often minimal. Most importantly, many hosts already do write-back caching, hence the benefit of having write-back at the disk cache is minimized. Also, some critical applications disable write-caching in order to ensure integrity constraints.

Table 3 shows our experiences with this issue for four benchmarks. Postmark [13] and TPC-

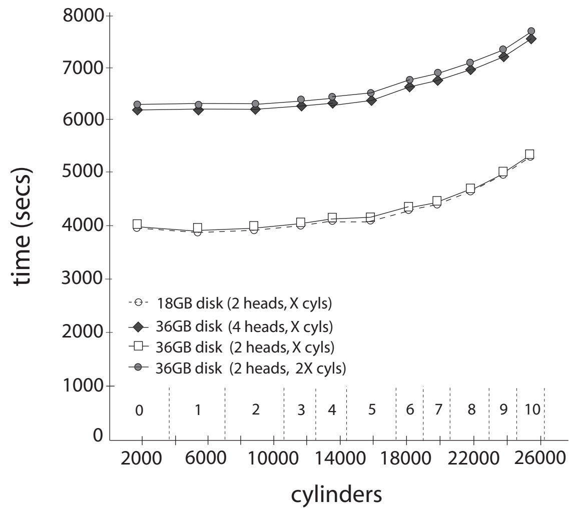
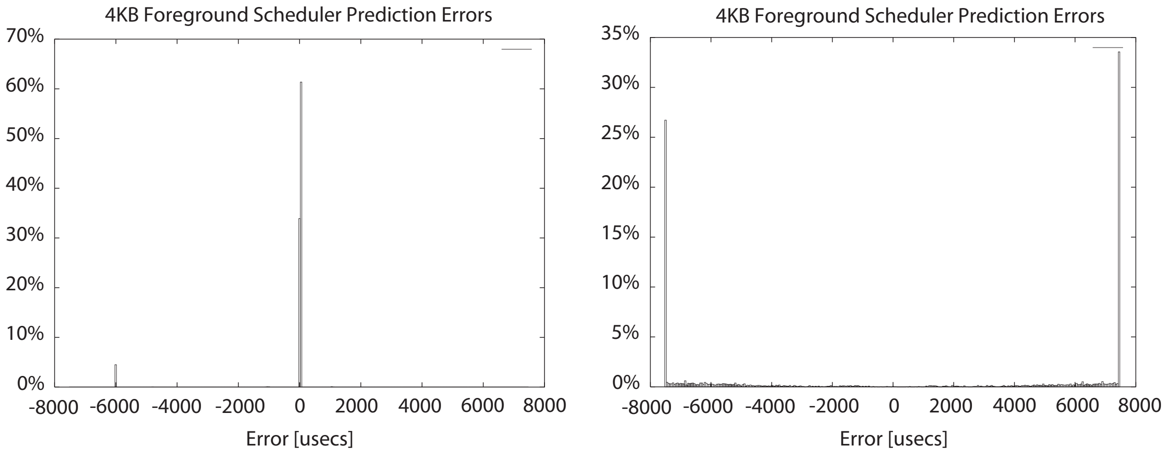


Figure 12: **A comparison of the algorithm efficiencies for different disks.** This diagram illustrates a simulated comparison of the main freeblock algorithm for an 18GB and 36GB disk. The 36GB disk is scaled in these three ways: by doubling the linear density of the disk, by doubling the number of disk heads, or by doubling the number of disk cylinders.

C [7] exhibit fairly random IO workload and it is unsurprising that the disk cache policy does not have a noticeable impact on their performance. When using soft-updates, SSH-build [22] and CP-DIFF (a program that copies two large traces on a partition and diffs them line by line) also do not experience any advantage or disadvantage when the disk cache policy changes. However, in the case when the partition does not use soft-updates, the performance of SSH-build degrades by more than 13% when the write policy is write-through. This is because synchronous metadata writes are able to return immediately after placing their buffers in the on-disk cache and the disk cache is effectively de-coupling the application writes and disk writes. CP-DIFF does not experience this degradation mainly because the large number of IOs when the traces are copied to the partition keeps the disk busy with doing writes (because the write buffer continuously fills itself) and the effect of the write-back cache is reduced.

In our current outside-the-disk implementation, the prediction accuracies are 90–99% accurate for the disks we are using. The scheduler uses dynamic conservative factors when appropriate to increase the prediction accuracy when it detects that mispredictions have been made. If the mispredictions persist, the freeblock subsystem does not attempt to schedule background tasks. Also, we are working on extending DIXtrac to report on a variety of features relevant to freeblock scheduling, such as disk aggressiveness on prefetching when two requests are issued on the same track.

Complex scheduling decisions: Because of limited command queuing (mentioned above) the scheduler must never allow more then two requests at the disk. This restriction has the drawback that in some cases it may result in sub-optimal scheduling decisions. Consider the case when three requests A, B and C come at the same time (approximately) at the device driver. Using FreeBSD's original scheduler they are all sent to disk and the disk starts with A, but then has the option to choose which of B or C is closest to the disk head when A completed. Thus the disk does SPTF scheduling. When using our scheduler, A and B are sent to the disk and C is left in the foreground



(a) **Prediction accuracy with write-through policy**

(b) **Prediction accuracy with write-back policy**

Figure 13: **Outside-the-disk scheduler prediction accuracies.** These graphs show the error in prediction accuracy defined as *error = actual - predicted time* The outside-the-disk access predictions deteriorate if the on-disk write policy is write-back.

pool. Effectively, we have a FIFO-like behavior, where we cannot choose whether B or C are closest to the disk head when A completes, since we have to do B next. This inefficiency could be solved by waiting until the disk head is close to A and then sending B down to the disk. But, this solution would require a timer be kept, similar to the one that is used to detect short and long idle times. This timer would have to be very precise and wake up the necessary code that sends B at the disk at exactly the right time.

Logical-to-physical mappings:: The set of disk makes and models with which we have worked utilize a broad variety of algorithms for mapping blocks to sectors. The result, in our code base, has been increasingly general functions for extracting model parameters and translating between logical and physical, which results in significant computational cost. Profiling suggests that specializing the system for any one disk type would provide substantial performance improvements. Such a trade-off is probably appropriate for some appliance vendors (e.g., file servers or TiVo boxes).

5 Summary

This paper describes a complete outside-the-disk freeblock subsystem, implemented in FreeBSD. It introduces new space- and time-efficient algorithms to utilize available idle time and rotational latency gaps in order to make freeblock scheduling useful in practice. The result is an infrastructure that efficiently provides steady disk access rates to background applications, across a range of foreground usage patterns.

6 Future work

This paper provides a blueprint for an efficient design of a practical outside-the-disk freeblock subsystem. Although our work illustrates that an implementation of such a system is indeed feasible, it also points out the difficulties and loss in efficiency when doing so outside-the-disk.

Most such difficulties would be almost entirely eliminated if the freeblock subsystem resided within the disk's firmware. Implementing freeblock scheduling inside the disk would also have the benefit of increasing the free bandwidth by more than 35%, as reported in [15]. Unfortunately, our implementation utilizes 2-8% of a 1 GHz CPU and 3-10MB of memory, which are hefty numbers for a disk drive. But, we made several implementation choices based on expedience and the resource-richness of host platforms. We believe that the CPU costs would go down by at least a factor of 2–4 with "in-the-disk" disk modeling support, which would have none of the generality overheads of our libraries. The memory requirements can also be reduced by more than an order of magnitude by restricting the numbers of concurrent tasks and sessions, which would in turn reduce CPU costs as well. We believe freeblock subsystem support is a valuable and viable feature for future disk drives.

Acknowledgements

We thank Vinod Das Krishnan, Steve Muckle, and Brian Railing for assisting with porting of freeblock scheduling code into FreeBSD. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This work is partially funded by the National Science Foundation, via grants #CCR-0326453 and #CCR-0113660.

References

[1] Mohamed Aboutabl, Ashok Agrawala, and Jean-Dominique 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] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Jospeh M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: making the fast case common. *Workshop on Input/Output in Parallel and Distributed Systems* (Atlanta, GA, May, 1999), pages 10–22. ACM Press, 1999.

[3] Paul 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] 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* (Florence, Italy, 07–11 June 1999), pages 400–405. IEEE, 1999.

[5] John S. Bucy and Gregory R. Ganger. *The DiskSim simulation environment version 3.0 reference manual*. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[6] 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.

[7] Transactional Processing Performance Council. TPC Benchmark C. Number Revision 5.1.0, 2002.

[8] 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.

[9] Zoran Dimitrijević, Raju Rangaswami, and Edward Chang. Design and implementation of semi-preemptible IO. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 145–158. USENIX Association, 2003.

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

[11] Robert Y. Hou, Jai Menon, and Yale N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. *Hawaii International Conference on Systems Sciences*, January 1993.

[12] David M. Jacobson and John 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] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[14] 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.

[15] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 275–288. USENIX Association, 2002.

[16] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik 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.

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

[18] Jeffrey C. Mogul. A better update policy. *Summer USENIX Technical Conference* (Boston, MA), pages 99–111, 6–10 June 1994.

[19] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993.

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

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

[22] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA, 18–23 June 2000), pages 71–84, 2000.

[23] 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, June 1998). Published as *Performance Evaluation Review*, **26**(1):44–55, 1998.

[24] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggreggate 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.

[25] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A framework for building unobtrusive disk maintenance applications. *To appear in the 3rd USENIX Conference on File and Storage Technologies (FAST '04). Also filed as Carnegie Mellon University Technical Report CMU-CS-03-192*, 2004.

[26] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 1–11. Usenix Association, 14–17 November 1994.

[27] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

[28] Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. Configuring and scheduling an eager-writing disk array for a transaction processing workload. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 289–304. USENIX Association, 2002.