A framework for building unobtrusive disk maintenance applications

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

October 2003
CMU-CS-03-192

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes a programming model and system support for clean construction of disk maintenance applications. Such applications expose the disk activity to be done, and then process completed requests as they are reported. The system ensures that these applications make steady forward progress without competing for disk access with a system's primary applications. It opportunistically completes maintenance requests by using disk idle time and free-block scheduling. In this paper, three disk maintenance applications (backup, write-back cache destaging, and disk layout reorganization) are adapted to the system support and evaluated on a FreeBSD implementation. All are shown to successfully execute in busy systems with minimal (e.g., <2%) impact on foreground disk performance. In fact, by modifying FreeBSD's cache to write dirty blocks for free, the average read cache miss response time is decreased by 15–30%.

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

There are many disk maintenance activities that are required for robust system operation and, yet, have loose time constraints. Such "background" activities need to complete within a reasonable amount of time, but are generally intended to occur during otherwise idle time so as to not interfere with higher-priority application progress. Examples include write-back cache flushing, defragmentation, backup, integrity checking, virus scanning, report generation, tamper detection, and index generation.

Current systems use a variety of ad hoc approaches for such activities. Most trickle small amounts of work into the storage subsystem, either periodically or when an idle period is detected.When sufficient idle time is not available, these activities either compete with foreground requests or are not completed. More importantly, trickling work into a storage subsystem wastes significant disk scheduling opportunities—it restricts the scheduler to only considering a small subset of externally-chosen requests at externally-chosen points in time. Most background activities need to read or write substantial portions of the disk, but do not have particular ordering requirements. As a result, some implementors try hard to initiate the right requests at the right times, introducing substantial complexity but, usually, only minor improvement.

This paper describes an alternate approach, wherein background activities are exposed to the storage subsystem so that it can schedule associated disk accesses opportunistically. With the storage subsystem explicitly supporting priorities, background applications can safely expose work and trust that it will not interfere with foreground activity. Doing so allows the scheduler to use freeblock scheduling and idle disk time to complete background disk accesses in the most device-efficient manner. Freeblock scheduling [24] predicts rotational latency delays and tries to fill them with media transfers for background tasks. As the set of desired disk locations grows, so does the ability of a freeblock scheduler to utilize such latency delays. The same is true for non-intrusive use of short periods of idle time. Combining rotational latency gaps with short and long periods of idle time, programs designed to work with storage-determined ordering can make consistent progress, without affecting foreground access times, across a wide range of workloads and levels of activity.

This paper describes a framework for background disk activities, including application programming interfaces (APIs) and support for them in FreeBSD. In-kernel and system call APIs allow background applications to register "freeblock tasks." Our *freeblock subsystem* replaces the generic SCSI driver's disk scheduler, utilizing both freeblock scheduling and any idle time to opportunistically complete freeblock requests. The APIs are explicitly asynchronous, and they encourage implementors to expose as much background work as possible. For example, dynamic buffer management allows freeblock tasks to register a desire to read more disk space than fits in main memory. Just-in-time locking avoids excessive holding of buffers, since freeblock writes may be pending for a long time. Rate control avoids memory exhaustion and wasted disk scheduling efforts for applications with low priority on the CPU.

We describe the conversion of three disk maintenance tasks to use of this infrastructure: scanning of disk contents for backup, flushing of write-back caches, and reorganizing disk layouts. Well-managed systems perform periodic backups, preferably without interfering with foreground activity. Backup is a great match for our framework, often reading large fractions of the disk; a "physical" backup does so without interpreting the file system structures and can be made order-agnostic. We implemented such a physical backup application that uses the freeblock subsystem to read disk blocks. Physical backup of a snapshot that covers more than 70% of an always-busy 18GB disk can be completed in a little over one hour with less than 2% impact on a foreground workload.

Almost all file servers and disk array controllers use write-back caching to achieve acceptable performance. Once updates are decoupled from application progress, they become a background activity appropriate for our framework. In our evaluations, we find that approximately 80% of the cache flushes can usually be eliminated, even when there is no idle time, reducing the average disk read response time by 15-25%. For low read-write ratios (e.g., 1:3–1:1), only 30–55% of the flushes are eliminated, but the read response time reductions are still 15–30%. Interestingly, when emulating a non-volatile cache, which eliminates FreeBSD's 30-second limit on time before write-back, almost all flushes can be eliminated, improving read response times by almost 50%.

Over time, allocated storage space becomes fragmented, creating a desire for defragmentation. Also, there have been many proposals for periodically reorganizing disk layouts to reduce future access times. Both require that disk blocks be shuffled to conform to a new, preferred layout. In our evaluations, we show that using our framework it is possible to reorganize the layouts quickly and with minimal impact on the foreground workloads.

This paper makes four main contributions: First, it describes in-kernel and system call APIs for background disk maintenance tasks. Second, it describes and evaluates three real examples using these APIs to cleanly and effectively function across a range of foreground usage models; each example and evaluation is interesting in and of itself. Third, it describes a general infrastructure for supporting these APIs. Fourth, it describes memory management and rate control practices necessary for robust operation.

The remainder of this paper is organized as follows. Section 2 discusses disk maintenance tasks, freeblock scheduling, and related work. Section 3 describes in-kernel and application-level APIs for background disk tasks. Section 4 describes three disk maintenance applications and how they use the APIs. Section 5 briefly describes the freeblock subsystem and its integration into FreeBSD. Section 6 evaluates how well the three applications work when using the framework.

2 Background and related work

There are many disk maintenance activities that need to eventually complete, but that ideally progress without affecting the performance of foreground activity. This section describes how such activities are commonly implemented, how a freeblock subsystem can help, and related work.

Characteristics and approaches: 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. As a result, one common approach is to simply postpone such activities until expected off hours; for example, desktop backups are usually scheduled for late nights (or, early in the morning for CS researchers). For less sporadically-used systems, however, the lower priority must be handled in another way.

Another common approach is to spread background requests over time so as to reduce interference with foreground work; for example, some caches flush a fraction of the dirty blocks each second to reduce penalties associated with periodic full cache flushes [9]. More aggressive implementations explicitly identify periods of idle time and use them to service background work. Of course, identifying idle times requires effort—the background activity must be invoked in the system's critical path—and assumptions about any pro-active storage-internal functions. When using a detected idle period, background activities usually provide only a few requests at a time to the storage subsystem in order to avoid having a lengthy queue when the next foreground request arrives. This is necessary because current storage systems provide little-to-no support for request priorities or preemption.

By providing only a few requests at a time, these implementations rob the disk scheduler of opportunities to reduce positioning times. In fact, disk maintenance applications usually need to access many disk locations and many could be quite flexible in their operation ordering. Some implementors attempt to recapture at least a portion of the lost efficiency by providing requests expected to be fast; for example, a disk array reconstruction task can, after a foreground request completes, generate background requests for locations near the recent foreground request rather than near the most recent background request [17]. Such tricks can provide marginal gains, but still lose out on much of the opportunity and often increase complexity by breaking abstraction boundaries between the application and the disk.

Freeblock scheduling: Since disk platters rotate continuously, a given sector will reach the disk head at a given time independent of what the disk head is doing until that time. Freeblock scheduling [24] consists of squeezing background media transfers into foreground rotational latencies. A freeblock scheduler predicts how much rotational latency would occur before the next foreground media transfer and inserts additional media transfers, while still leaving time for the disk head to reach 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 time available for the additional media transfers, but not completely eliminating it.

Freeblock scheduling, as originally proposed, combines nicely with idle time usage to provide disk bandwidth to background tasks across a wide range of foreground usage patterns. In addition to detecting and using lengthy idle time periods, low-level scheduling can allow short, sporadic idle periods to be used with minimal penalty. Throughout this paper, we use the term *freeblock scheduling* to refer to this more complete scheduling combination.

Freeblock scheduling is a good match for many disk maintenance activities, which desire large numbers of disk blocks without requiring a predetermined order of access. Properly implemented, such activities can provide much freedom to a scheduler that opportunistically matches rotational latency gaps and idle time bursts to desired background transfers.

Related work: Lumb et al. [24] first evaluated the use of rotational latency gaps for background work and evaluated the system via simulation. The simulations 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 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 different groups [23, 37] 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 [23] or writes in eager writing disk arrays [37]. 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 by describing a general programming framework for background disk tasks and evaluating several uses of it.

Dimitrijević et al. [12] 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.

Several interfaces have been devised to allow application writers to expose asynchronous and order-independent access patterns to storage systems. Dynamic sets [32], disk-directed I/O [21], and River [4] all provide such interfaces. We borrow from these, and asynchronous networking interfaces like sockets, for the APIs described in the next section.

3 Background disk I/O interfaces

To work well with freeblock scheduling, applications must be designed explicitly for asynchronous I/O and minimal ordering requirements. An application should describe to the freeblock subsystem sets of disk locations that they want to read or write. Internally, when it can, the freeblock subsystem inserts requests into the sequence sent to the disk. After each desired location is accessed, in whatever order the low-level disk scheduler chooses, the freeblock subsystem informs the application and provides any data read.

This section describes two generic application APIs for background activities. The first is an in-kernel API intended to be the lowest interface before requests are sent to the storage device. The second API specifies system calls that allow user-level applications to tap into a freeblock subsystem. These APIs provide a clean mechanism for registering background disk requests and processing them as they complete. Applications written to these interfaces work well across a range of foreground usage patterns, from always-busy to frequently-idle. Both APIs talk in terms of logical block numbers (LBNs) within a storage logical unit (LUN); consequences of this choice are discussed in Section 3.4.

3.1 In-kernel API

Table 1 shows the in-kernel API calls for our freeblock scheduling subsystem. It includes calls for registering read and write freeblock tasks, for aborting and promoting registered tasks, and for suspending and resuming registered tasks. As a part of the high-level device driver, there is one instance of the freeblock scheduler per device in the system; the standard driver call switch mechanism disambiguates which device is intended. This section explains important characteristics of the API.

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.[1] Sessions allow applications to suspend, resume, and set priorities (values between 1 and 100, with a default of 20) 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 report subtask completions. The freeblock scheduling subsystem promises to read or write each identified disk

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

Function Name	Arguments	Description
fb_open	*priority, callback_fn, getbuffer_fn*	Open a freeblock session (ret: *session_id*)
fb_close	*session_id*	Close a freeblock session
fb_read	*session_id, addr_range, blksize, callback_param*	Register a freeblock read task
fb_write	*session_id, addr_range, blksize, callback_param*	Register a freeblock write task
fb_abort	*session_id, addr_range*	Abort parts of registered tasks
fb_promote	*session_id, addr_range*	Promote parts of registered tasks
fb_suspend	*session_id*	Suspend scheduling of a session's tasks
fb_resume	*session_id*	Resume scheduling of a session's tasks
**(callback_fn)*	*session_id, addr, buffer, flags, callback_param*	Report that part of task completed
**(getbuffer_fn)*	*session_id, addr, callback_param*	Get memory address for selected write

Table 1: **In-kernel interface to the freeblock subsystem.** *fb_open* and *fb_close* open and close a freeblock session for an application. Tasks can be added to a session until the application closes it. *fb_read* and *fb_write* register one or more freeblock tasks. *fb_abort* and *fb_promote* are applied to previously registered tasks, to either cancel pending freeblock tasks or convert them to foreground requests. *fb_suspend* and *fb_resume* disable and enable scheduling for all tasks of the specified session. **(callback_fn)* is called by the freeblock subsystem to report data availability (or just completion) of a read (or write) task. The freeblock subsystem reclaims the buffer when **(fb_callback)* returns, meaning that the callee must either process the data immediately or copy it to another location before returning control. When a write subtask is selected by the scheduler, **(getbuffer_fn)* is called to get the source memory address.

location once and to call *callback_fn* when freeblock requests complete. On the last callback for a given session, the *flags* value is set to the appropriate value indicating completion.

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. Having only a portion of a database page, for example, may be insufficient to process the records therein. The *blksize* value must be a multiple of the LBN size (usually 512 bytes). In practice, it must be small to allow scheduling flexibility; for modern disks, *blksize* values above 64 KB substantially reduce the effectiveness of freeblock scheduling when there is no idle time [33].

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. In this latter case, the freeblock request is not generated and that part of the freeblock task waits for a future opportunity.

The original reason for *getbuffer_fn* was to avoid long-held locks on buffers associated with registered freeblock write tasks. Commonly, file systems and database systems lock cache blocks for which disk writes are outstanding in order to prevent them from being updated while being DMA'd to storage. With freeblock scheduling, writes can be waiting in the freeblock task list for a long time; such locks could easily be a system bottleneck. The *getbuffer_fn* callback allows the lock to be acquired at the last moment and held only for the duration of the actual disk write. For example, the free write-backs described in Section 4.2 actually hurt performance when they do not utilize this functionality. Since adding it to the API, we have found that the *getbuffer_fn* function cleanly supports other uses. For example, it enables a form of eager writing [35, 37]: one can register freeblock write tasks for a collection of unallocated disk locations and bind unwritten new blocks to locations in *getbuffer_fn*. The disk write then occurs for free and the relevant metadata can be updated with the resulting location.

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.

3.2 Application-level API

The application-level API mirrors the in-kernel API, with a system call for each *fb_xxx* function call. The main differences are in notification and memory management. Because the kernel must protect itself from misbehaving applications, the simple callback mechanisms of the low-level API are not feasible in most systems. Instead, a socket-like interface is used for both.

As with the in-kernel API, an application begins by calling *sys_fb_open* to get a *session_id*. It can then register freeblock tasks within the session. For each block read or written via these tasks, a completion record is inserted into buffers associated with the session. Applications get records from these buffers via the one new call: *sys_fb_getrecord (buffer)*; each call copies one record into the specified application *buffer*. Each record contains the *session*, *addr* and *flags* fields from *callback_fn* in the in-kernel API, as well as the data in the case of freeblock reads. Like with sockets, the *sys_fb_getrecord* call can be used for both blocking and polling programming styles.[2] A *timeout* parameter in the *sys_fb_getrecord* function dictates how long the application will wait if no completion record is currently available. A value of 0 will return immediately (polling), and a value of -1 will wait indefinitely.

3.3 Consistency model

Freeblock tasks may have long durations; for example, a background disk scan can take over an hour. Therefore, a clear consistency model is needed for overlapping concurrent freeblock and foreground requests.

Like most low-level storage interfaces, our APIs opt for maximum scheduling flexibility by enforcing a minimalistic semantic with three rules. First, no ordering guarantees are enforced among pending tasks, whether they overlap or not. Second, data returned from a read should have been on the disk media at some point before being returned. Third, a block write can be reported

[2]Our experiences indicate that full integration with existing system call mechanisms would be appropriate. Specifically, using the standard file descriptor mechanism would allow integrated use of `select()` with sockets, from which this interface borrows many characteristics. For example, given such integration, an application could cleanly wait for any of a set of sockets and freeblock sessions to have made progress.

complete when it is on disk or when a concurrent write to the same disk location completes; the latter case is rationalized by the fact that the non-written blocks could have been put on the disk just before the ones actually put there.

Given these semantics, a freeblock scheduler can coalesce some overlapping tasks. Of course, data fetched from media can be replicated in memory and passed to all concurrent readers. In addition, completion of a write task to location A allows completion of all pending reads or writes to A because the newly written data will be the current on-disk data once the write completes. As a result, a write is given preference when a set of overlapping reads and writes are pending; a read could be done before the write, but doing so is unnecessary given the consistency model.

3.4 Consequences of LBN-based interface

The freeblock scheduling APIs described interact with driver-level scheduling in terms of LBNs. This simplifies implementation of the scheduler and of low-level disk maintenance tasks, such as RAID scrubbing and physical backup. But, most utilities that access structured storage (e.g., files or databases) must coordinate in some way with the software components that provide that structure. For example, consider a file-based backup application. It could read a directory and register freeblock tasks to fetch the files in it, but it will not know whether any given file is deleted and its inode reallocated between the task being registered and the inode eventually being read from disk. If this happens, the application will backup the new file under the old name. Worse problems can arise when directory or indirect blocks are reallocated for file data.

Three options exist for maintenance tasks that interact with structured storage. First, the task could coordinate explicitly with the file system or database system. Such coordination can be straightforward for integrated activities, such as segment cleaning in an LFS [27] or index generation in a database system. Second, the task could insist that the file system or database system be temporarily halted, such as by `unmount`ing the file system. Although heavy-handed, a system with many file systems could have individual ones halted and processed one-by-one while the others continue to operate on the storage devices. Third, the task could take advantage of an increasingly common mechanism in storage systems: the snapshot [16]. A snapshot provides an instance of a dataset as it was at a point in history, which is useful for backup [10] and remote replication [26]. Since the contents of a snapshot remain static, update problems are not an issue for tasks using the freeblock scheduling APIs. In addition to traditional backup tasks, snapshots offer a convenient loose coordination mechanism for disk maintenance tasks like integrity checking, virus scanning, report generation, tamper detection, and garbage collection. Section 4 shows an example of how a backup application interacts with the snapshot system and the freeblock subsystem.

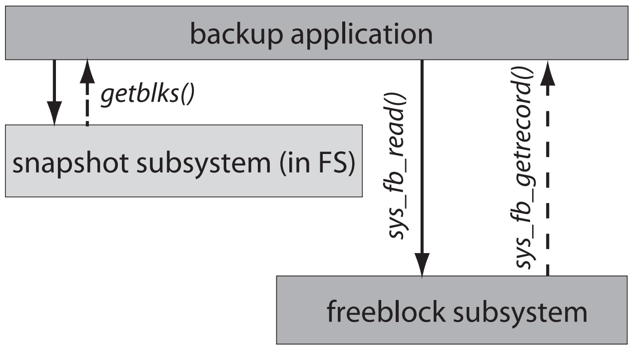


Figure 1: **Snapshot-based backup application.** The backup application interacts with the snapshot subsystem to learn which blocks comprise the snapshot in question. It uses the freeblock subsystem to read these blocks from disk.

4 Example applications

Many disk maintenance applications can be converted to the programming model embodied in our APIs. This section describes the conversion of three such applications that are evaluated in Section 6.

4.1 Snapshot-based backup

Most systems are periodically backed-up to ensure that the data stored is not lost by user error or system corruption. In general, it is accepted that either the system will be otherwise idle during the backup time or the backup will have some performance impact on foreground activity [13, 15, 19].

Backup strategies fall into two categories: logical and physical backup. Logical backup is a file-based strategy. It first needs to interpret the file system's metadata and find the files that need to be backed-up. The files are then stored to the backup media in a canonical representation that can be restored at a later time. The advantages of logical backup include the ability to restore specific files and to backup only live data. Physical backup is a block-based strategy. Physical backup does not interpret the file structure that it is backing up. Uninterpreted raw blocks are copied from one media to another. The main advantages of physical backup are its simplicity and scalability. In particular, physical backup can achieve much higher throughput while consuming less CPU [19].

Physical backup fits well with our programming model. No ordering among blocks is required. Instead, blocks are copied from one device to another as they are read. The blocks could be written to the backup media out of order (and reorganized during restore), or a staging device could be used to reorder before writing to tape.

Physical backup can take advantage of snapshots, which are on-line read-only copies of a volume of storage. Snapshots allow consistent backup from an active on-line system. Snapshots were added to the 5.x release of FreeBSD [25].

Our backup application uses FreeBSD's snapshot infrastructure and our system call API. No changes are required to the FreeBSD snapshot implementation. After a snapshot is taken, the backup application interacts with the snapshot subsystem as shown in Figure 1. First, it gets the list of blocks that belong to the snapshot file. Then, the backup application registers freeblock tasks, via *sys_fb_read*, to read them. It interactively calls *sys_fb_getrecord* to wait for reads to complete and get the data. Each successfully read block is sent to the back-up destination, together with its address. The backup application can also be used to create a replica by writing each block directly to the corresponding LBN on the destination LUN.

FreeBSD's approach to handling modifications to blocks "owned" by a snapshot creates an additional complexity for the backup application. A snapshot implementation can do one of two things when a block is modified. In the first option ("application-copy-on-write"), a new location is chosen for the updated blocks and the snapshot map stays unchanged. Network Appliance's WAFL file system, for example, uses this method [19]. In the second option ("snapshot-copy-on-write"), the original data is copied to a newly allocated block and the snapshot map is modified. FreeBSD uses this second option to avoid disrupting carefully chosen disk assignments. In the evaluation section, we explore the effects of both methods on the backup application.

To handle FreeBSD's snapshot-copy-on-write, the backup application needs to check with the snapshot system whether each returned block still has the original desired contents. If not, a new freeblock task to read the relocated block is registered. This procedure continues until all original blocks have been read. Note that we could have changed the snapshot subsystem to automatically abort and re-register tasks for modified blocks, but our intention is to show that the backup application works well even with an unmodified snapshot system.

4.2 Buffer cache cleaner

Caches are part of all storage systems, and most are write-back in nature. Data blocks written to the cache are marked *dirty* and must eventually make their way to the storage device. In most operating systems, including FreeBSD, the cache manager promises applications that data written to the cache will propagate to persistent storage within a certain fixed window of time, usually 30 seconds. This persistence policy tries to bound the amount of lost work in the face of a system crash. In many file servers and disk array controllers, cache persistence is not a concern because they utilize battery-backed RAM or NVRAM. But, dirty buffers must still be written to storage devices to make room in the cache for new data. Although these systems do not necessarily need a persistence policy, they still need a cache write-back replacement policy.

Cache write-back is a good application for a freeblock subsystem. In most cases, there are no ordering requirements and no immediate-term timeline requirements for dirty blocks. Until a persistence policy or cache space exhaustion is triggered, they are background activities that should not interfere with foreground disk accesses (i.e., cache misses).

We modified FreeBSD's cache manager to utilize our in-kernel API. It registers all dirty buffers to be written for free through the use of the *fb_write* call. Cache blocks are not locked when the writes are registered; when its *getbuffer_fn* is called by the freeblock subsystem, the cache manager returns **`NULL`** if the lock is not free. When its *callback_fn* is called, the cache manager marks the associated block as clean. If the freeblock subsystem still has not written a buffer for free when the cache manager decides it must be written (as a consequence of the cache replacement or persistence policies), then the cache manager can convert the associated write to a foreground request via *fb_promote*. If a dirty buffer dies in cache, for example because it is part of a deleted file, the task registered to flush it to disk can be aborted through *fb_abort*.

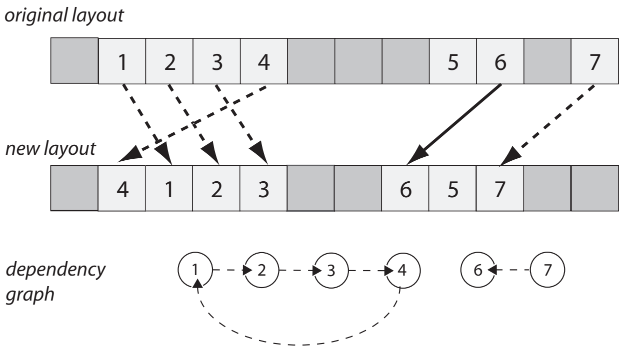


Figure 2: **Sample dependency graph for disk layout reorganization.** This diagram illustrates the dependency graph that results from changing the disk layout. The gray boxes represent empty physical locations. The white boxes present physical locations that have been mapped to a particular LBN (given by the number on the box). Dashed arrows present dependencies whereas solid lines show movements that do not have any dependencies.

4.3 Layout reorganizer

Disk access times are usually dominated by positioning times. Various layout reorganization heuristics have been developed to reduce access times. For example, blocks or cylinders may be rearranged in an organ pipe fashion, as in [29]. Files or blocks may be replicated so each read can access the closest replica, as in [22, 36]. Blocks may be rearranged to make disk accesses more sequential, as in [3, 29].

Layout reorganization is a background activity that can be made to fit our programming model. But, doing so requires that the implementer think differently about the problem. In the traditional approach, most work focuses on planning an optimal movement pattern. Because a freeblock subsystem is opportunistic, extensive forward planning is not useful, since one cannot predict when freeblock opportunities will be available.

Planning movement patterns is a focus because there are dependencies between block movements. If a block is to be moved to a location that currently contains live data, the live data must first be read and either moved or buffered. Although no block can directly depend on more than one other block, dependency chains can be arbitrarily deep. Figure 2 illustrates an example of these dependencies.

Our layout reorganizer, illustrated in Figure 3, respects these dependencies, of course, but does so without trying to create a plan based on the dependency graph. Instead, it attempts to maximize the freeblock subsystem's flexibility while ensuring that deadlock cannot occur. To do so, it uses an NVRAM staging area[3] and three distinct freeblock sessions. The pool of staging buffers is logically partitioned into two parts: writable and unwritable.

The "read-staged" session registers read tasks for all blocks that could not yet be written, due to a dependency; these blocks, when read, become part of the unwritable portion of the staging pool. The "read-writable" session registers read tasks for all blocks that can be immediately written (i.e., they have no dependencies) or that clear a dependency for a currently buffered block; when read, a read-writable block either becomes a writable buffer or becomes an unwritable buffer but

[3]Our experimental system does not actually have NVRAM. Instead, the layout reorganizer just allocates a block of memory and pretends it is non-volatile. This emulates how the reorganizer might work in many modern file servers and disk array controllers

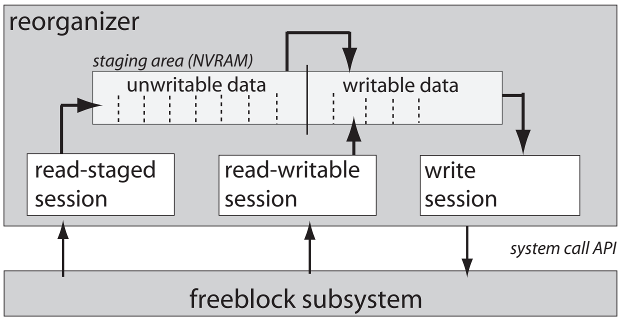


Figure 3: **Layout reorganizer architecture.** This diagram illustrates the design of the layout reorganizer implemented using our framework. The read-staged session manages blocks whose dependencies have not yet been solved. The read-writable session manages all blocks that can be read because their dependencies have been solved. The write-session manages all block writes. All data is temporarily stored in the NVRAM staging area.

also converts an unwritable buffer to writable.

Whenever a completed read creates a writable buffer, the "write" session registers a write task for it. If a current read-staged task depended upon it, that task no longer has a dependency. Therefore, it is aborted (*sys_fb_abort*) and re-registered as a read-writable task. When a write completes, its buffer can be released from the staging area and reclaimed.

In order to avoid deadlocking, the reorganizer ensures that the number of unwritable blocks in the cache never exceeds a threshold percentage of the cache. If the number of unwritable buffers reaches the threshold, the reorganizer suspends (*sys_fb_suspend*) the read-staged session, allowing only the read-writable and write sessions to continue. Since read-writable read tasks are either immediately writable or will allow an in cache block to be immediately writable, reading them cannot consume the remaining staging space without allowing forward progress. When unwritable space falls below the threshold, because of writes and/or cleared dependencies, the read-staged session can be restarted.

The combined size of the unwritable and writable pools in the staging area is also limited. The reorganizer suspends both read sessions if the staging area fills, until blocks in the cache are written out to disk. When buffer space becomes available, the reorganizer uses the *sys_fb_resume* call to resume the scheduling of read tasks.

5 The freeblock subsystem

This section briefly describes the freeblock subsystem infrastructure implemented in FreeBSD to experiment with our background applications. This infrastructure supports all the background disk I/O APIs described in Section 3. Details and evaluation of this infrastructure are available in [33].

5.1 Architecture and integration

Figure 4 illustrates the major components of our freeblock subsystem. The background scheduler exports the in-kernel API, and a *system call translator* component translates the application-level API calls to in-kernel calls. This section describes these pieces and their integration into

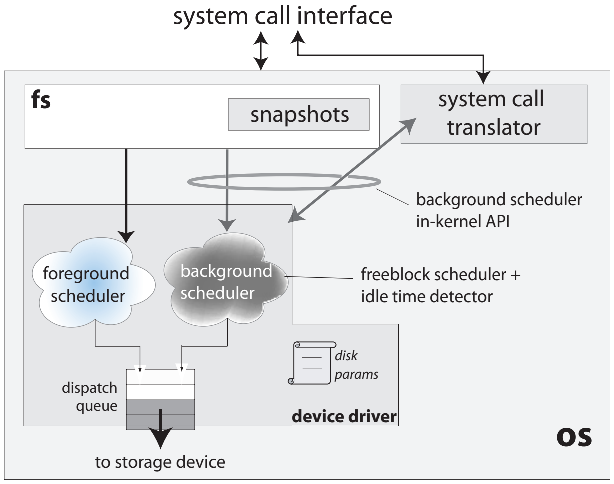


Figure 4: **Freeblock subsystem components.**

FreeBSD.

Foreground and background schedulers: Our scheduling infrastructure replaces FreeBSD's C-LOOK scheduler. The foreground scheduler uses Shortest-Positioning-Time-First (SPTF), and the background scheduler uses freeblock scheduling (rotational latency gaps and any idle time). Both schedulers use common library functions, which are much like other recent software-only outside-the-disk SPTF models [1, 5, 6, 23, 31, 36], for modeling the disk to predict positioning times for requests. In our case we use DIXtrac [30] to extract the disk parameters shown in Figure 4.

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, *buf_queue*, which is the dispatch queue in Figure 4. 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 inserted 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.

Freeblock system call translator: The system call translator implements the application-level API. Doing so consists of translating system calls to in-kernel calls and managing the flow of data between the freeblock subsystem and the user-level application. When a freeblock task completes, the translator's *callback_fn* appends a record to the associated session's buffers and, if the buffers were empty, awakens any waiting application processes. When the freeblock subsystem reads data faster than the application processes it, the buffers associated with the session fill up and flow control is needed. When this happens, the translator uses the *fb_suspend* call, suspending subsequent freeblock requests for the tasks associated with the given session. When the application fetches records and thereby clears space, the translator uses *fb_resume* to re-enable the associated freeblock tasks. When an application exits or calls *sys_fb_close*, the translator clears all state maintained by the freeblock system on behalf of the application's session(s).

5.2 Background scheduler algorithms

The background scheduler includes algorithms for utilizing otherwise wasted rotational latency gaps and for detecting and using disk idle time.

Rotational latency usage: Recall that during busy disk periods rotational latency gaps can be carefully filled with background media transfers. Our freeblock subsystem uses algorithms similar to those described by Lumb et al. [23, 24], modified to use less non-idle CPU time and to support fairness and priorities among freeblock sessions.

The search for suitable background transfers proceeds in two phases. The first phase checks only a few tracks for potential background transfers adding an insignificant amount of computation ($<<5\%$) to a busy CPU. The second phase only runs when the CPU is otherwise in the idle loop. It searches all other options in order to refine the best choice found until the request needs to be sent.

Prior algorithms greedily scheduled freeblock requests, assuming all were equal. As shown in Section 6.6, this can lead to poor behavior when freeblock sessions are mixed. In particular, full disk scans can starve other sessions. We introduce fairness, as well as support for priorities, using a simple form of lottery scheduling [34]. The initial tickets allocated to each session are proportional to its assigned priority.

The lottery determines both which pending tasks are considered, since there is limited CPU time for searching, and which viable option found is selected. During the first phase, four cylinders worth of pending tasks from the winning session are considered: the two closest to the source and the two closest to the destination. 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.

Idle time detection and usage: Previous research [14, 28] reports that most idle periods are a few milliseconds in length and that long idle time periods come in multi-second durations. Our freeblock subsystem utilizes both.

When the last pending foreground request completes, the scheduler considers pending freeblock reads on the same track. Such data can be read and cached in the device driver with minimal impact on foreground access patterns, because no mechanical delays are induced and no disk prefetching is lost.

Borrowing from prior work [14], a simple threshold (of 20ms) is used to identify likely long idle periods. For each quanta of a long idle period, a session is selected via the lottery. Pending tasks of the winning session are scheduled, starting with the most difficult to service using rotational latency gaps: those near the innermost and outermost cylinders.

Algorithm summary: Our outside-the-disk freeblock scheduler has the same "imperfect knowledge and control" limitations described by Lumb et al. [23], and thereby loses about 35% of the potential free bandwidth. An implementation embedded in a disk drive could be expected to provide correspondingly higher free bandwidth to applications. The introduction of conservative CPU usage further reduces free bandwidth utilization by 5–10%. In our evaluations we show that this bandwidth is adequate for most background applications. More detailed description of the data structures and algorithms, and evaluation of them, are available in [33].

6 Evaluation

This section evaluates how effectively the framework supports the three background applications.

6.1 Experimental setup

All experiments are run on a system with a dual 1GHz Pentium III, 384MB of main memory, an Intel 440BX chipset with a 33MHz, 32bit PCI bus, and an Adaptec AHA-2940 Ultra2Wide SCSI controller. Unless otherwise stated, the experiments use a Seagate Cheetah 36ES disk drive with a capacity of 18GB. Two implementations of the freeblock subsystem are used: one in the FreeBSD device driver, one in user-level Linux. The user-level Linux implementation can either do direct SCSI reads and writes or communicate with a simulated storage device implemented by DiskSim [7]. All implementations use the same scheduling core and conservatism factors used in the FreeBSD implementation.

Three benchmarks are used throughout the evaluation section. The **synthetic benchmark** is a multi-threaded program that continuously issues small (4KB-8KB) read and write I/Os to disk, with a read-write ratio of 2:1, keeping two requests at the disk queue at all times.

The **TPC-C benchmark** simulates an on-line transaction processing database workload, where each transaction consists of a few read-modify-write operations to a small number of records [11]. The disk locations of these records are pseudo-random. We ran TPC-C on the Shore database storage manager [8]. It includes a volume manager for storing individual pages, a buffer pool manager, lock and transaction managers, and a recovery subsystem. We configured Shore and TPC-C to use 8KB pages, a 64MB page buffer pool, 50 warehouses (covering approximately 70% of the Seagate disk's capacity) and 10 clients-per warehouse. The Shore volume is a file stored in FreeBSD's FFS file system. Thus, an I/O generated by Shore goes through the file system buffer cache. Performance of a TPC-C benchmark is measured in TPC-C transactions completed per minute (TpmC)

The **Postmark benchmark** was designed to measure the performance of a file system used for electronic mail, netnews and web-based services [20]. It creates a large number of small files and performs a specified number of transactions on them. Each transaction consists of two subtransactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 200,000 transactions on 800,000 files in 10,000 directories. File sizes range from 10KB to 20KB. The biases are Postmark's default ones (read/append=5, create/delete=5).

6.2 Freeblock subsystem effectiveness

This section briefly evaluates the freeblock subsystem's effectiveness. Figure 5 shows the efficiency of the freeblock subsystem as a function of disk utilization.

The synthetic benchmark is used. A background disk scan registers a freeblock task to read every block of the disk. The *callback_fn* is instrumented so that no part of this task is ever satisfied

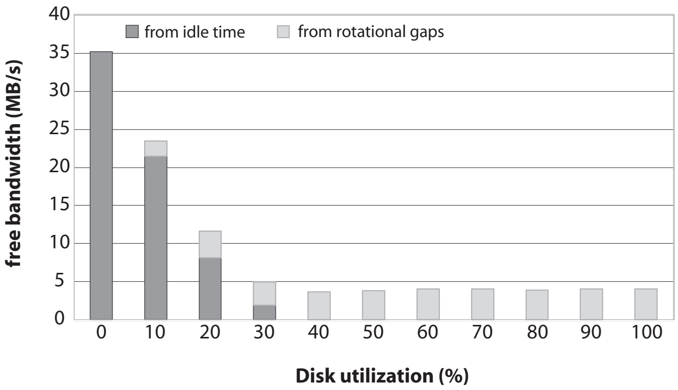


Figure 5: **Freeblock subsystem efficiency.** This diagram illustrates the free bandwidth extracted for a background disk scan as a function of the disk's utilization. When the foreground workload is light, idle time is the main source of free bandwidth. When the foreground workload intensifies, the free bandwidth comes from rotational gaps

(whenever blocks are read, they are re-registered to be read again). This guarantees that at all times there is a constant amount of blocks wanted to be read in the background (in this case the whole disk). The synthetic benchmark is modified slightly so that the number of I/Os per second can be varied, and the request arrival rate is uniformly distributed.

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 40-60% utilized. This is due to the scheduling algorithms used. Regardless of utilization, foreground requests are affected by less than 2%. For a full evaluation of the freeblock infrastructure and algorithms, please refer to [33].

6.3 Snapshot-based backup

This section evaluates our backup application described in section 4.1. We evaluate both the application-copy-on-write (application-COW) and snapshot-copy-on-write (snapshot-COW) strategies in the FreeBSD kernel. When application-COW is used, all subsequent modifications to a block that the snapshot claims are sent to a new location. When snapshot-COW is used, all subsequent modifications go to the original location of the block and the snapshot system makes a private copy of the block which it then claims. The native snapshot implementation in FreeBSD supports only snapshot-COW; we instrumented the kernel so that we could evaluate application-COW as well.

Figure 6 shows the performance of our backup application when sharing the system with the three foreground benchmarks. The table beneath the graph shows that the impact of the concurrent backup on foreground performance is less than 2%. Under the synthetic benchmark, the backup is performed faster than under Postmark or TPC-C. This is so because the synthetic benchmark's requests are uniformly distributed around the disk, maximizing the scheduler's opportunities. The backup is slightly faster under TPC-C than under Postmark for several reasons. First, due to frequent file deletes, a large percentage of dirty blocks die in cache. Thus less foreground I/O is generated and fewer freeblock opportunities arise. Although not enough idle time is detected, the disk is not fully utilized (see Figure 5). Second, the cache manager successfully coalesces many small dirty buffers for Postmark and thus issues fewer larger I/Os to the device, which reduces the

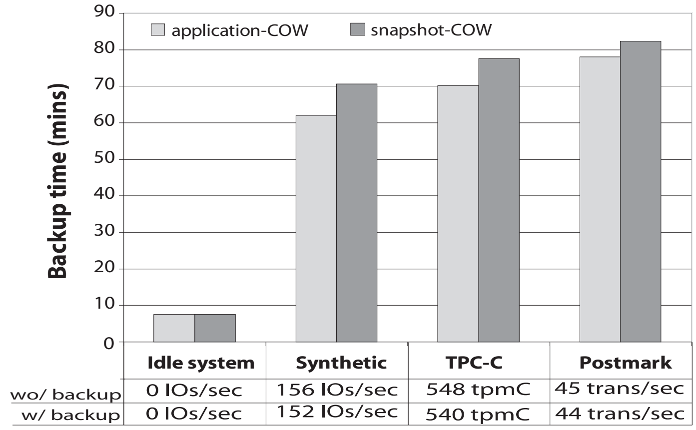


Figure 6: **Snapshot-based backup efficiency.** This diagram illustrates the efficiency of the backup application when backing up 70% of the Cheetah 36ES disk (18GB). The foreground workload is affected less than 2% during the background backup as a result of access time mispredictions that result from the outside-the-disk implementation of freeblock scheduling.

effectiveness of freeblock scheduling further.

In the idle time case the streaming bandwidth is about 35MB,[4] and the backup completes in little over 8 minutes.

The graph also shows that the application-COW results in more efficient use of free bandwidth. This is because snapshot-COW wastes some of the bandwidth reading blocks that have been modified. Snapshot-COW then needs to re-register reads for the new locations of those blocks. The overall effect however, is less then a 15% increase in the time to complete the backup.

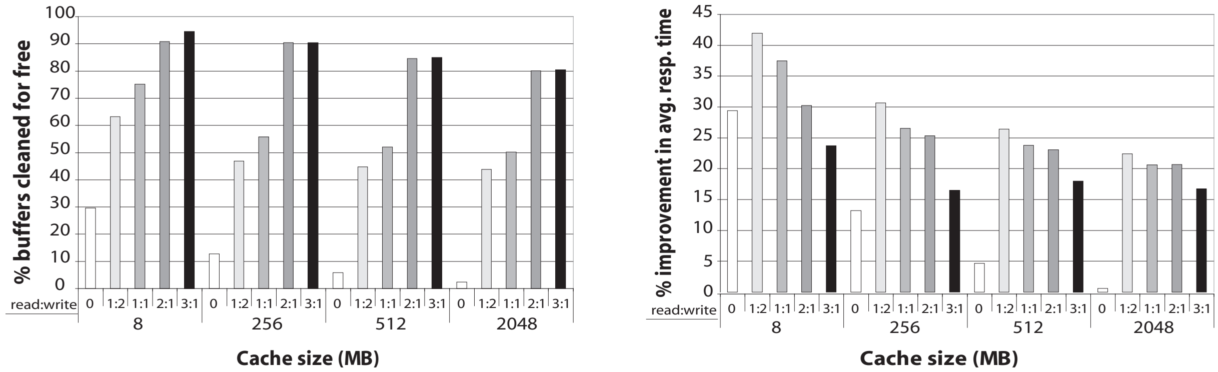
6.4 Buffer cache cleaner

We measure the efficiency of the buffer cache cleaner designed using our framework using both controlled experiments (using the Linux user-level implementation with a simulated cache and direct SCSI reads and writes on the Seagate disk) and the implementation in FreeBSD. The controlled experiments are used to understand the relationship between the efficiency of the cache cleaner and the size of the cache, the workload presented and the replacement and persistence policies. The metrics we are interested in are the percentage of dirty blocks cleaned for free and the reduction in average response time of other requests as a result of the free write-backs. In all buffer cache experiments, the idle-time detector does not detect enough idle time to be helpful.

For our controlled experiments, we use a version of the synthetic benchmark. We vary the read-write ratio and the simulated cache size while keeping the size of the requests the same (4KB-8KB).

Figure 7 shows the efficiency of the cleaner and its impact on the overall response time as a function of the workload's read-write ratio and the cache size. The replacement policy is least-recently used (LRU), and the persistence policy guarantees that no dirty buffer will stay dirty for longer than 30 seconds. High and low water-marks are used to address space exhaustion: whenever the number of dirty buffers in the cache hits the high water-mark, the cache manager cleans up as

[4] The reported streaming bandwidth of the disk is 40MB/s. But, due to head switch delays when changing tracks, the observed streaming bandwidth is about 35MB/s.



(a) **Efficiency of the cache cleaner with LRU and syncer daemon**

(b) **Average response time improvement from free write-backs**

Figure 7: **LRU replacement policy with syncer daemon.** These graphs illustrate the efficiency of freeblock scheduling and the impact it makes on the average response time of a system with LRU replacement policy and with a syncer daemon that guarantees no dirty block will stay dirty for longer than 30 seconds. The x-axis contains the cache size and the read-write ratio. A read-write ratio of 0 means that all requests are writes.

many buffers as needed until the low water-mark is reached. Mimicking the notation used by the FreeBSD's cache manager, a *syncer daemon* implements the persistence policy, and a *buffer daemon* implements the logic that checks the high and low water-marks.

Several observations can be made from Figure 7. First, as the read-write ratio increases, a larger percentage of the dirty buffers can be cleaned for free. This is because having more reads in the system means that more requests are hitting the disk immediately and thus more freeblock opportunities are created. Writes do not go to disk immediately because of write-back caching. Instead, they go to disk as a result of the syncer's work or buffer daemon's work. In both cases, they go to disk in large bursts. Hence, the foreground scheduler (using SPTF) does a good job in scheduling, reducing freeblock scheduling's chances of finding rotational gaps to use.

Second, as the read-write ratio increases (beyond 1:2), the impact of free cleaning on the average response time decreases. This is a direct consequence of the decreasing number of writes (and hence, dirty buffers) in the system.

Third, the efficiency of the freeblock subsystem slightly decreases with increasing cache size. The reason is that every time the syncer or buffer daemons wake up, they have a larger amount of dirty buffers to flush. Again, the foreground scheduler reduces the freeblock scheduler's chances of finding rotational gaps to use.

Figure 8 examines the efficiency of the cache cleaner and its impact on the average response time under different replacement policies. The cache size is kept fixed (512MB) and the read-write ratio is 1:1. Apart from LRU, two other replacement policies are used. The SPTF-Evict policy is similar to LRU, but instead of replacing dirty entries in an LRU fashion, the entries closest to the disk head position are replaced first. The FREE-CLEAN (FC) policy chooses to replace a *clean* entry that has been recently cleaned for free (if none exists, it reverts to LRU). By replacing a clean entry from the cache, FREE-CLEAN attempts to let the remaining dirty buffers stay a little longer in the system so that they may be written out for free.

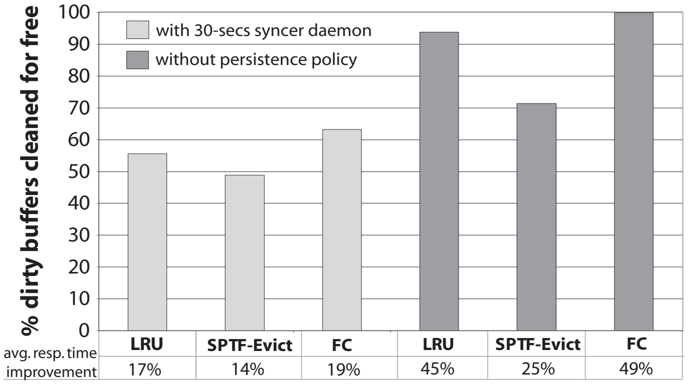


Figure 8: **Comparison of replacement and persistence policies.** These graphs illustrate the efficiency of the cache cleaner on a system under different replacement and persistence policies.

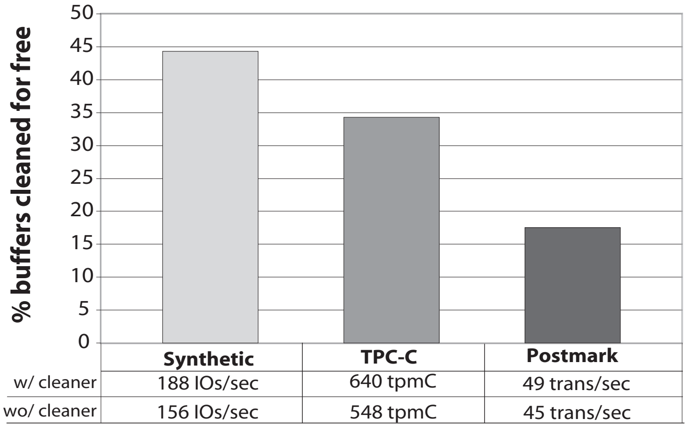


Figure 9: **Cache cleaner efficiency.** This diagram illustrates the efficiency of the FreeBSD cache cleaner under various workloads.

All three replacement policies are evaluated with, and without, a syncer daemon. A syncer daemon places a hard limit (30 seconds in our case) on the time the freeblock subsystem has to clean any dirty buffers for free. Hence, fewer buffers are cleaned for free under this policy, irrespective of the replacement policy used. However, as mentioned in Section 4.2 a cache comprised of non-volatile RAM does not need such a persistence policy.

The SPTF-Evict policy reduces the effectiveness of the freeblock subsystem most, thereby reducing its benefit to the average response time. This is because no write task can be satisfied during write I/Os that happen as a result of the buffer daemon (because the dirty buffer closest to the disk head is written first there are no other dirty buffers freeblock scheduling can squeeze in between foreground requests). Write tasks can still be satisfied during write I/Os that happen because of the syncer daemon. In the case when no syncer daemon is used, all writes happen due to the buffer daemon, hence dirty buffers can be cleaned for free only during foreground read requests.

The FC is the best policy as far as the cache cleaner is concerned. By leaving the dirty buffers in the cache a little longer, it can clean more of them for free. But, there could be a detrimental effect on cache hit rate, and the cache cleaning benefit observed is quite small.

Figure 9 illustrates the efficiency of the real cache cleaner, implemented as part of FreeBSD's

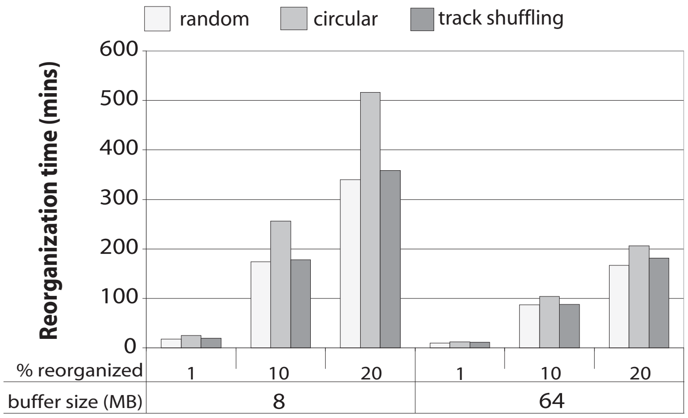


Figure 10: **Layout reorganizer efficiency.** This diagram illustrates the efficiency of the layout reorganizer for different reorganization heuristics.

cache manager. At most, 3/4 of the system's 384MB of RAM are devoted to the I/O buffering subsystem. FreeBSD uses a syncer daemon, which places a 30 limit on the time a buffer can stay dirty in the system. The read-write ratio of the synthetic benchmark is 1:1, the observed read-write ratio of TPC-C is approximately 1:1, whereas Postmark has a read:write ratio of approximately 1:3. In all three cases, a large percentage of the dirty buffers are cleaned for free and the respective improvement in application throughput ranges from 9 to 20%. Postmark lags behind the other benchmarks for the same reasons it lagged in the backup evaluation: write-back clustering and deletion of dirty blocks.

6.5 Layout reorganizer

To evaluate the effectiveness of our reorganizer, we performed a variety of controlled experiments. The foreground workload is the synthetic benchmark mentioned. In all cases, the system is 100% utilized. To avoid corruption of the FFS file system in FreeBSD, the experiments are run in the user-level Linux environment, with DiskSim as the backend storage device. In all experiments, the base unit the reorganizer is interested in moving at any time is 8KB (this is specified by the *blksize* parameter).

Three different reorganization actions are explored.

Random reorganization: This method chooses random 8KB blocks on the disk to move them to another random location. Few blocks have dependencies using this method.

Circular random reorganization: The method creates a list of unique random 8KB blocks, and then moves each block to the location of the next block in the list. This way we have the longest dependency chain possible; one including every block to be reorganized.

Track shuffling: This method is similar to the random block reorganization method above, but whole tracks instead of blocks are shuffled.

We evaluated each method reorganizing from 1% to 20% of the disk. The research on reorganization techniques has shown that this range is generally the most effective amount of the disk to reorganize [2, 18, 29]. The results are shown in Figure 10. Tests with more dependencies, like circular, take longer than those with few dependencies, and benefit more from an increase in buffer size. Track shuffling performs slightly worse than the random movement because while

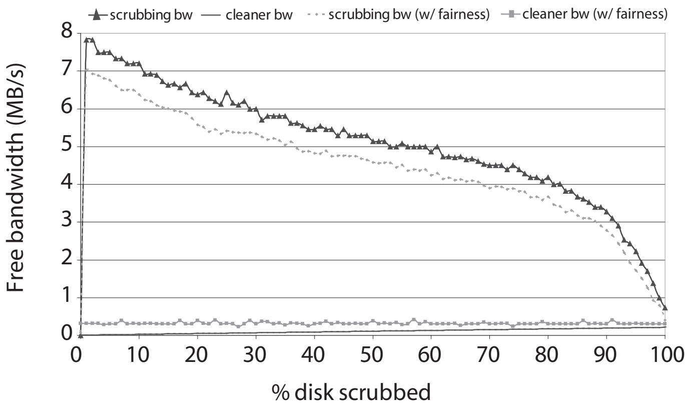


Figure 11: **Disk scrubbing and cache cleaning.** This figure shows two background applications, disk scrubbing and cache cleaning, in a system with and without fairness.

track aligned operations are efficient, the higher locality is less ideal for freeblocks, as described in [33].

The results are encouraging, showing that up to 20% of the disk can be reorganized in a few hours on a fully busy disk.

6.6 Application fairness and priorities

This section briefly evaluates the fairness of the scheduling algorithms. Two applications compete for the free bandwidth: a simple disk scrubber and the cache cleaner evaluated above. The disk scrubber simply tries to read all blocks of the disk once, without worrying about consistency issues (hence it doesn't use the snapshot system). The experiment is run until the disk scrubber has read all blocks of the 18GB Seagate disk.

The bandwidth dedicated to the scrubber and cache cleaner applications is measured. In the original case, the freeblock scheduler's fairness mechanisms are disabled and the scheduling algorithms lean toward a greedy approach. In the fair system, lottery scheduling makes sure that both applications are treated fairly. The priorities assigned to both applications are equal **DEFAULT_PRIORITY**. The cache size is fixed to 512MB, the replacement policy is LRU, and the persistence policy is implemented using the 30-sec syncer daemon. The read-write ratio of the foreground workload is 1:1.

Figure 11 shows the distribution of bandwidth with and without fairness. The bandwidth given to the cache cleaner increases from almost nothing to about 0.3MB/s when priorities are used. This bandwidth is very close to 0.34MB/s, which is the bandwidth the cache cleaner would get if it were the only background application in the system. The bandwidth of the scrubber, on the other hand, falls by a little more than the gained bandwidth of the cache cleaner. This 2-5% loss in efficiency can be attributed to the scheduler's decision to treat the cache cleaner in a fair manner, thereby spending an equal time searching for opportunities that satisfy tasks of that application. These opportunities are smaller when compared to the opportunities of the scrubber.

7 Summary

This paper describes a programming framework for developing background disk maintenance applications. With several case studies we show that such applications can be adapted to this framework effectively. A freeblock subsystem can provide disk access to these applications, using freeblock scheduling and idle time with minimal impact on the foreground workload.

We are currently exploring extensions to the freeblock subsystem to allow background applications to specify by how much they are willing to impact the performance of foreground applications (currently, the subsystem tries to enforce a strict priority). We are also building a simple library on top of the system call APIs to allow applications to use the usual file-based *read(), write()* calls for background access to snapshots.

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.

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] Sedat Akyurek and Kenneth Salem. *Adaptive block rearrangement*. CS–TR–2854. University of Maryland, February 1992.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. *ACM SIGMOD International Conference on Management of Data* (Minneapolis, MN, 24–27 May 1994). Published as *SIGMOD Record*, **23**(2):383–394, 1994.

[9] 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.

[10] Ann L. Chervenak, Vivekanand Vellanki, and Zachary Kurmas. Protecting file systems: a survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference* (March 1998), 1998.

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

[12] 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–2 April 2003). USENIX Association, 2003.

[13] Ben Duhl, Steven Halladay, and Prakash Mankekar. Disk backup performance considerations. *International Conference on Management and Performance Evaluation of Computer Systems*, pages 646–663, 1984.

[14] 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.

[15] Russell J. Green, Alasdair C. Baird, and J. Christopher. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, **8**(2):32–45, 1996.

[16] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[17] 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.

[18] Windsor Hsu. *Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance*. PhD thesis, published as UCB/CSD–03–1223. University of California at Berkeley, January 2003.

[19] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 239–249. ACM, Winter 1998.

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

[21] 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.

[22] Sai-Lai Lo. *Ivy: a study on replicating data for performance improvment*. TR HPL-CSP-90-48. Hewlett Packard, December 1990.

[23] 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.

[24] 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.

[25] Marshall Kirk McKusick. Running 'fsck' in the background. *BSDCon Conference* (San Francisco, CA, 11–14 February 2002), 2002.

[26] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 117–129. USENIX Association, 2002.

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

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

[29] Chris Ruemmler and John Wilkes. *Disk Shuffling*. Technical report HPL-91-156. Hewlett-Packard Company, Palo Alto, CA, October 1991.

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

[31] 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.

[32] 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.

[33] Eno Thereska. *A framework for implementing background storage applications using freeblock scheduling*. Masters Thesis. Carnegie Mellon University. August 2003.

[34] 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.

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

[36] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.

[37] 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.