Active Block I/O Scheduling System (ABISS)

Benno van den Brink
benno.van.den.brink@philips.com

Werner Almesberger
werner@almesberger.net

August 12, 2004

Abstract

The Active Block I/O Scheduling System (ABISS) is an extension of the hard-disk storage subsystem of Linux, whose main purpose is to provide a guaranteed reading and writing bit rate to applications.

1 Introduction

The availability of inexpensive mainstream IDE Hard Disk Drives (HDD) has allowed the use of these disk drives in home and mobile audio-visual (A/V) applications. This fast-increasing storage capacity has created a new class of devices like HDD video recorders, personal audio players etc. Because the number of streams that has to be read or written to disk is usually limited to one or two, streaming from and to a hard disk is currently not an issue in these devices.

Currently a clear trend is visible in which CE devices will become interconnected through home networks in the near future. Devices like these will need to be able to serve multiple data streams, while providing a 'soft real-time' service. This sharing should be efficient because - even more than e.g. in the traditional PC environment - CE devices often have to meet other constraints like low power consumption, noise-free operation, minimum hardware cost, etc. Resource sharing can be accomplished by either making the applications aware of each other, or by making the system aware of the applications.

In this paper we will present the results of work done on the hard-disk storage subsystem of Linux, resulting in the Active Block I/O Scheduling System (ABISS). The main purpose of ABISS is to make the system application-aware by either providing a guaranteed reading and writing bit rate to any application that asks for it or denying access when the system is fully committed. Apart from these guaranteed real-time (RT) streams, our solution also provides priorities for best-effort (BE) disk traffic.

The system consists of a framework that is included in the kernel, with a policy and coordination unit implemented in user space. This approach ensures separation between the kernel infrastructure (the framework) and the policies (e.g. admission control) in user space.

The kernel part consists of our own elevator and a new 'read scheduler', communicating with a user-space daemon. The elevator implements the multiple priorities of the streams and the read scheduler is responsible for timely preloading and buffering of data. Apart from the elevator and read scheduler, some minor modifications were made to file system drivers.

ABISS works from similar premises as RTFS [1], but puts less emphasis on tight control of low-level operations, and more on convergence with current Linux kernel development.

In section 2 we will give an overview of the system, in section 3 the implementation is described in more detail. Some measurements will be presented in section 4.

The ABISS project is hosted at `http://abiss.sourceforge.net/`

2 Overall architecture

In this section, we describe the role of the individual components that make up the ABISS system, and how they interact.

Figure 1 shows the data path when reading from disk: the application issues requests to VFS, which translates them to requests for the disk blocks that comprise the corresponding data pages.[1] The block IO subsystem then queues these requests, and feeds them to the device driver, which retrieves the data from disk.

1. When using the term "request" in this paper, we usually refer to such a block IO request. The kernel data structure describing such a request is aptly named `struct request`.

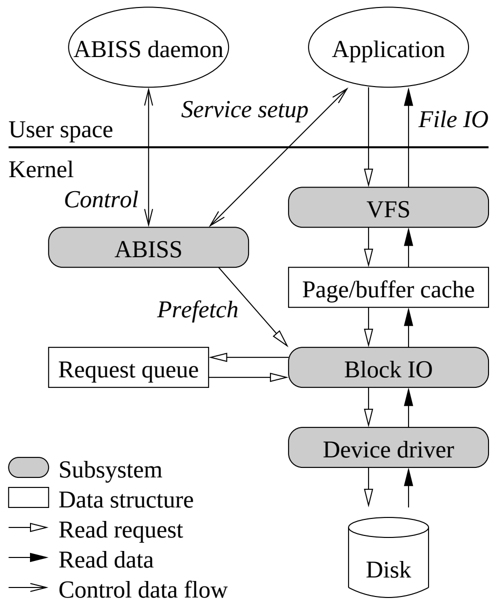


Figure 1: The ABISS real-time service enhances regular file IO by prefetching data such that the application never has to wait for disk accesses.

When an application requests a real-time *service* (see below) from ABISS, the so-called scheduler component of ABISS prefetches data into the page cache such that it will already be in memory when the application reads it.

ABISS consists of a generic framework and modules that implement specific services. The kernel part is assisted by a user-space daemon, which oversees system-wide resource use, and makes policy decisions. In this paper, we only discuss the *real-time* service available in the default configuration. In the future, other services may be added.

2.1 Application and service model

With ABISS, an application can specify for each *open file*[2] in qualitative and quantitative terms how it expects accesses to be handled. This description defines how the application will behave, and what service it expects from the operating system. ABISS then decides whether system resources allow it to provide this service, and whether the application is entitled to it. If yes, ABISS makes the necessary arrangements, and indicates to the application that the service is available.

The application then accesses the file in accordance with the profile it has specified in its request, and receives the agreed upon service in return. While doing so, the application may give ABISS indications of this use, e.g. by informing it about the current position in the file.

When the application closes the file, the service is automatically terminated. The usual rules for sharing of open files apply, e.g. if an application `forks`, the same service is used jointly by both processes then sharing the file.

2.2 Service definition

In the rest of this paper, we will focus on the guaranteed real-time service. This service only applies to reading.

From the application's point of view, the real-time service is characterized by a rate (r) and a buffer size (b). The application sets the *playout point* to mark the location after which it performs accesses. As long as the playout point moves at rate r or less, accesses to up to b bytes after the playout point will be served from memory. If the application moves the playout point faster, the range shrinks according to the excess rate, and grows again towards b if the application slows down.

For a formal definition, if we consider reading a file as a sequence of n single-byte accesses, with the i-th access at location a_i, at time t_i, and with the playout point set to p_i, the operating system then guarantees that all accesses are served from memory, as long as the following conditions are met for all i, j in $1, \ldots, n$ with $t_i < t_j$:

$$\begin{aligned} p_i &\leq p_j < p_i + b + r(t_j - t_i) \\ p_j &\leq a_j < b + \min(p_j, p_i + r(t_j - t_i)) \end{aligned}$$

The infrastructure can also be used to implement a prioritized best-effort service without guarantees. Such a service would ensure that, on average and when measured over a sufficiently long interval, a reader that has always at least one request pending, will experience better latency and throughput, than any reader using a lower priority.

2.3 API

ABISS does not require any changes in the way applications read and write files. Also memory-mapped file access is fully supported.

However, in some cases, the ABISS scheduler (see below) needs further help from the appli-

2. We use the term "open file" to refer to what POSIX [2] calls an "open file description", i.e. the object a file descriptor points to. For one file (represented in the kernel by `struct inode`), there can be many open files (represented by `struct file`).

cation. For example, when reading a file, particularly if memory-mapped, the kernel cannot reliably determine the exact location of the application's playout point.[3] Therefore, the application needs to explicitly send this information.

Applications use `ioctl`s for all ABISS-related communication. For convenience, there is also a library of wrapper functions, providing a higher-level interface.

2.4 User-space daemon

When an application requests a service from ABISS, the request is examined and refined in several steps. This is shown in figure 2. First, the request is checked for formal validity. For example, it may specify a service that is different from what has been configured on the file system in question, or the set of parameters may be incomplete. Most of these problems are detected by the general framework of ABISS in the kernel, even before calling the scheduler.

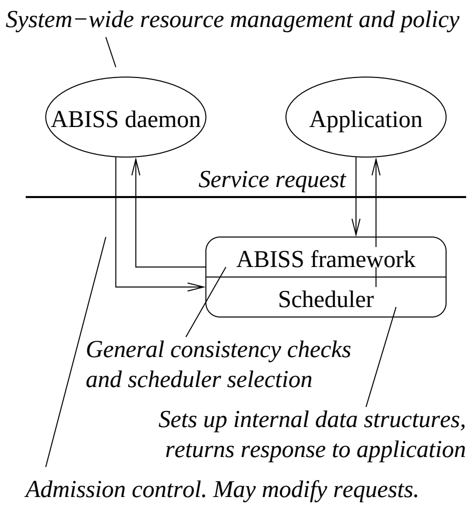

System−wide resource management and policy
ABISS daemon
Application
Service request
ABISS framework
Scheduler
General consistency checks and scheduler selection
Sets up internal data structures, returns response to application
Admission control. May modify requests.

Figure 2: When processing a service request, the kernel delegates the admission control decision to the ABISS user-space daemon.

Then, the request is sent to the ABISS daemon. This daemon keeps track of system-wide resource utilization, and decides whether the system is capable of providing the requested additional service. This decision can also include policy, such as quotas assigned to individual users, and other access control considerations.[4]

The ABISS daemon can also modify the request and add new parameters. The request is then passed back to the kernel, and – if it was accepted – used by the scheduler to actually implement the corresponding service.

Communication between the ABISS daemon and the kernel is message-based, and uses a miscellaneous device.

2.5 Scheduler

The *scheduler* implements the time-related aspects of a service, and defines its properties. It is assisted by the user-space daemon described above.

Schedulers are modules in the ABISS system, and can be configured individually for each mounted file system. Each scheduler module may also implement several different services or operating modes. In the rest of this paper, we will focus on the "test" scheduler, which implements the real-time and prioritized best-effort services described in section 2.2.

When using the real-time service, the scheduler prefetches pages of the file the application is reading. This is similar to the *read-ahead* functionality the kernel normally provides, but uses the buffering requirements the application specified to prefetch pages such that the application will never try to read a page that has not been prefetched yet. Furthermore, the scheduler assigns a high priority to its read requests, so that the time until a page is read from disk becomes predictable. Priorities are implemented by the elevator, which is described below.

The scheduler limits the rate at which the application can read data at real-time priority to the rate the application specified when requesting the real-time service.

2.6 Playout buffer

When an application reads data from a file, this data is normally buffered in kernel space, and then transferred to user space as needed. The kernel also tries to read data ahead of time, so that the application does not have to wait for the actual disk access. Data is normally prefetched in multiples of one memory page (a page has typically a size of 4 kB).

For real-time reads, ABISS builds upon this concept, and prefetches data according to the rate at which the application will read it. This is illustrated in figure 3. Pages are still kept in the page cache, but in addition to this, they are

3. The playout point is the file location from which the application is currently reading. We discuss the playout point in more detail in section 3.2.

4. At the time of writing, policy is not yet implemented.

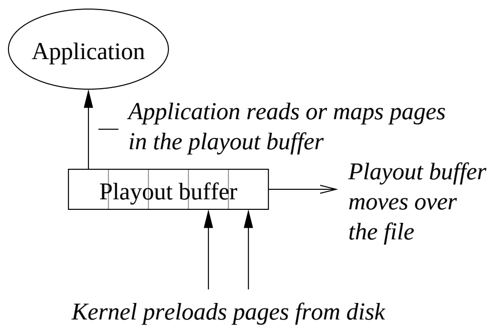

Application
Application reads or maps pages in the playout buffer
Playout buffer
Playout buffer moves over the file
Kernel preloads pages from disk

Figure 3: The kernel prefetches pages into the playout buffer, such that they are in memory when the applications retrieves them.

locked in memory while they are in the playout buffer.[5]

Besides the data rate, ABISS also takes into account buffering requirements of application and kernel. The data rate is specified by the application when requesting the real-time service. The application also specifies its own buffering requirements, for which it has to take into account the following factors:

- The amount of data the application will retrieve or access at once.
- The amount of time the application may be ahead or behind of the specified rate. This may be due to the way that the application is designed, but it may also be due to delays inflicted by the operating system.
- Any deviation from a sequential access pattern.

There are also various operating system and hardware properties that need to be taken into account when dimensioning the playout buffer. Their handling is transparent to the application, and explained in section 3.2.

2.7 Elevator

The *elevator*[6] orders disk IO requests in a way that minimizes movements of the disk drive head, and that also tries to ensure that no application monopolizes disk accesses. Linux allows configuration of the elevator at boot time, and, as an extension [3], also at run time.

ABISS has its own elevator that implements eight distinct priorities, as shown in figure 4. Requests at a lower priority are only served if there are no requests at a higher priority. Priorities are assigned by the ABISS scheduler, and

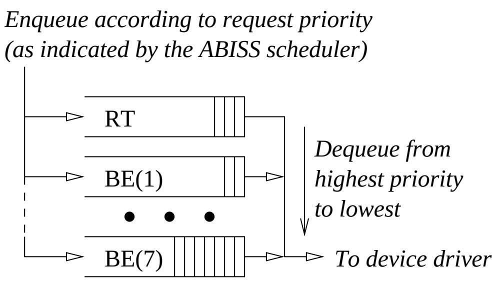

Enqueue according to request priority (as indicated by the ABISS scheduler)
RT
BE(1)
BE(7)
Dequeue from highest priority to lowest
To device driver

Figure 4: The ABISS elevator implements eight priorities: one for real-time requests (RT), and seven for best-effort (BE).

can be per page (for real-time) or per file. The elevator is discussed in more detail in section 3.6.

2.8 Scopes

Figure 5 illustrates for which areas of the system the individual parts of ABISS are responsible.

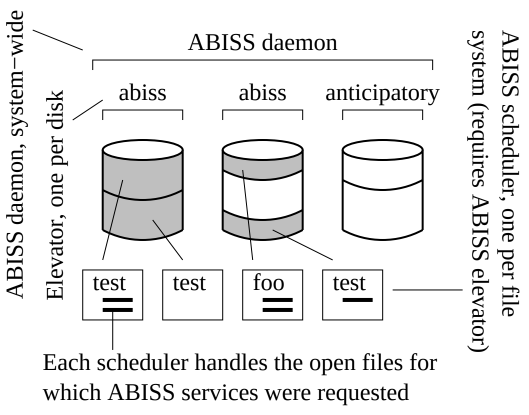

ABISS daemon, system−wide
Elevator, one per disk
ABISS daemon
abiss
abiss
anticipatory
test
test
foo
test
ABISS scheduler, one per file system (requires ABISS elevator)
Each scheduler handles the open files for which ABISS services were requested

Figure 5: Example of how the elements of ABISS are connected to distinct parts of a system.

The daemon oversees resource use in the whole system. Elevators are configured per disk device. If using ABISS on any part of a disk, the entire disk must therefore be handled by an instance of the ABISS elevator. ABISS schedulers can be chosen individually for each mounted file system. Finally, each scheduler takes care of all open files on that file system, which are serviced

5. Files read using direct IO are not buffered in the kernel, and can therefore not be combined with the ABISS real-time service.

6. On Linux, the elevator is frequently called the "IO scheduler". In this paper, we always use "elevator", to avoid confusing it with ABISS' scheduler, or the CPU scheduler.

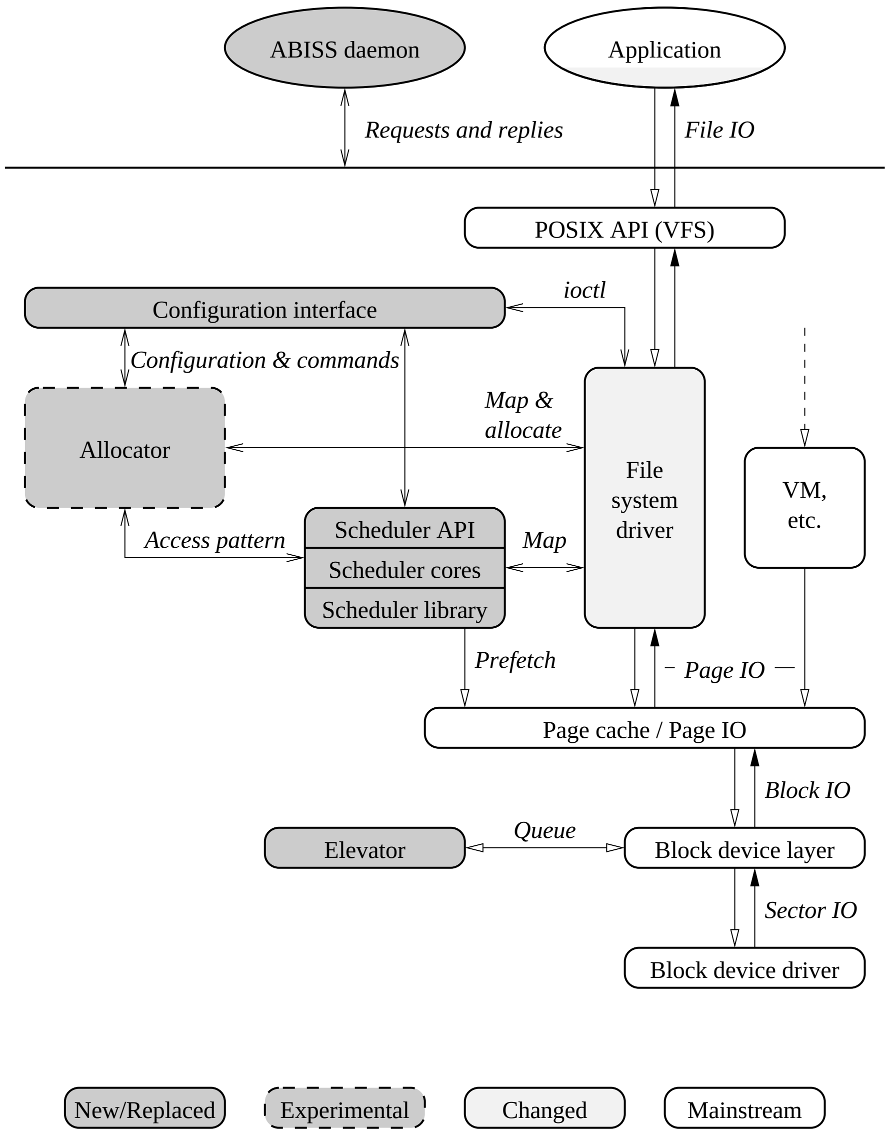
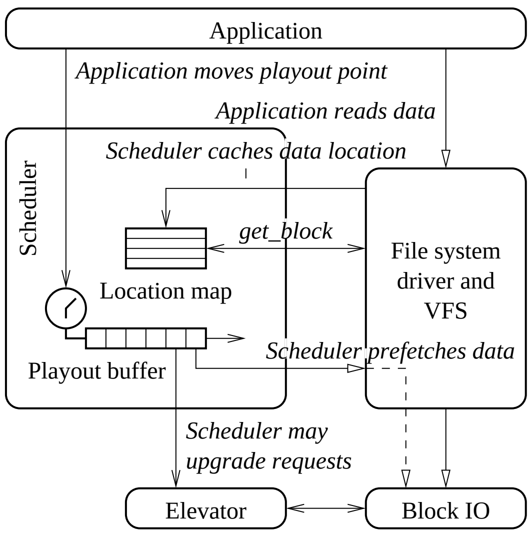
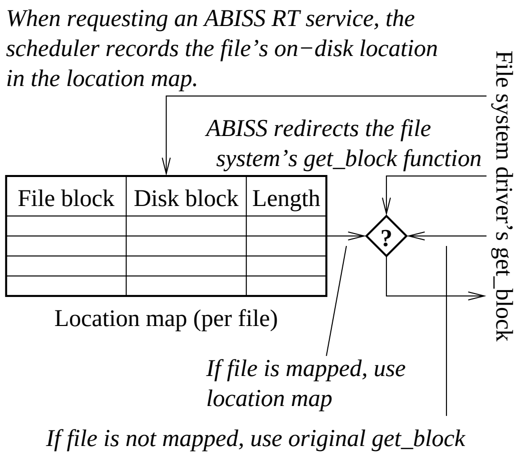


Figure 6: Overview of how ABISS interfaces with the existing Linux IO subsystem.



Application
Application moves playout point
Application reads data
Scheduler
Scheduler caches data location
get_block
Location map
File system driver and VFS
Playout buffer
Scheduler prefetches data
Scheduler may upgrade requests
Elevator
Block IO

Figure 7: The main components of the "test" scheduler are the location map, the playout buffer, and the prefetch logic.



When requesting an ABISS RT service, the scheduler records the file's on−disk location in the location map.
ABISS redirects the file system's get_block function
File block
Disk block
Length
?
File system driver's get_block
Location map (per file)
If file is mapped, use location map
If file is not mapped, use original get_block

Figure 8: The scheduler queries the file system driver for the on-disk location of file data, and then uses this location map to read the file without file system meta-data accesses.

by ABISS. Files on other file systems, and files for which no ABISS service has been requested, are not seen by the schedulers, and are handled with a default best-effort priority.

3 Implementation

Figure 6 shows the main components of the Linux IO subsystem and of ABISS. When an application reads or writes files, it uses the POSIX API and VFS to convey the operations to the file system driver. Then, the file system driver (through generic support functions not shown here) generates accesses to the page cache.

Data is transferred between the page cache and the disk through the block device layer: first, a block IO request (`struct bio`) is assembled, which is then sent to the block device layer, where it is turned into transfer requests for a number of disk sectors. These requests (`struct request`) are put into the request queue of the disk elevator. The disk device driver then picks such requests from the queue one after the other, and processes them.

The processing path for memory-mapped files is similar. The main difference is that the activity is triggered through the VM subsystem and goes only then to the file system driver.

In order to use ABISS with a file system, the file system driver needs to be changed. The changes mainly consist of adding a call to the ABISS ioctl function and looping the driver's `get_block` and `release` functions through ABISS.

ABISS also provides its own elevator, which implements priorities. This elevator has to be used by all devices on which file systems providing ABISS services are mounted.

Some small changes must be made to the application, to request an ABISS service, and to communicate with the scheduler.

The other parts of ABISS are completely new: the scheduler orchestrates IO operations such that the service goals are met. It is assisted in this by the ABISS daemon in user-space, which oversees global resource use and makes policy decisions.

The allocator is an experimental component for controlling write operations. This component is described in more detail in section 5.

3.1 Scheduler

As shown in figure 7, the "test" scheduler contains two major functional blocks:

- The "location map" caches the on-disk location of file data, and helps to avoid difficult to handle accesses to file system meta-data during real-time reading. There is one location map per `struct inode`.
- The playout buffer caches file data, as described in section 2.6. The scheduler does the prefetching and also controls the rate at which real-time operations occur. There is one playout buffer per `struct file`.

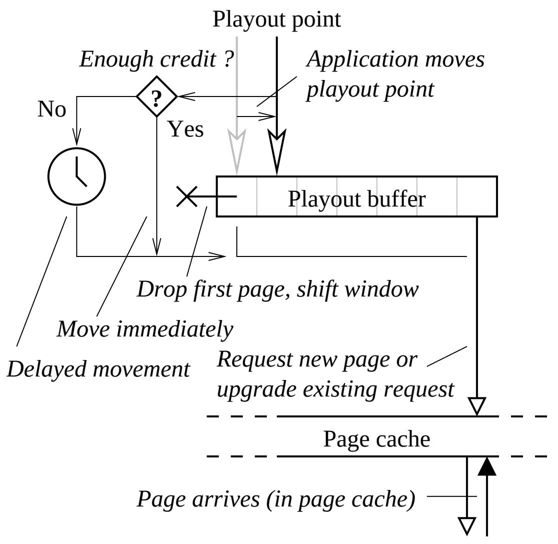

Playout point
Enough credit ?
Application moves playout point
No
?
Yes
Playout buffer
Drop first page, shift window
Move immediately
Delayed movement
Request new page or upgrade existing request
Page cache
Page arrives (in page cache)

Figure 9: Playout buffer movement is initiated by the application moving its playout point, and may happen either immediately, or when sufficient credit becomes available.

When preparing a file for real-time service, the scheduler first looks up the locations of all disk blocks occupied by file data, and stores these locations for later use, in a data structure called the "location map". That way, the file can later be read without accessing meta-data, which makes it easier to predict the activity resulting from this read operation. The use of the location map is shown in figure 8.

The file system driver's `get_block` function is changed such that it calls a function in the scheduler instead. This function checks if a location map is available for the file, and if so, looks up the block in question. If no location map is available, the original `get_block` function of the file system driver is called.

The location map is also updated when writing to the file. It is implemented as a red-black tree.

3.2 The playout buffer

The playout buffer is the heart of the scheduler, and also its most complex part. This section describes the process of moving it, and how it is dimensioned.

When a file is set up for ABISS service, the playout buffer is filled (at a best-effort priority, but as quickly as possible) before the real-time service begins.

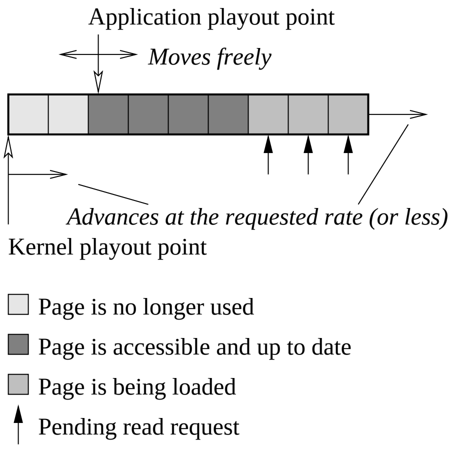

Application playout point
Moves freely
Advances at the requested rate (or less)
Kernel playout point
Page is no longer used
Page is accessible and up to date
Page is being loaded
Pending read request

Figure 10: Playout buffer movement is controlled by the position of two playout points, one from the application, and the other from the kernel.

Moving the playout buffer

Figure 9 illustrates how the playout buffer movements through the file: first, the application tells the scheduler to move the playout point by some number of bytes. The scheduler then checks if it should move the playout buffer (by one or more pages), and if there is enough *credit* available for this. The credit is a measure of how far the buffer can be moved at a given time. This concept is described in more detail below.

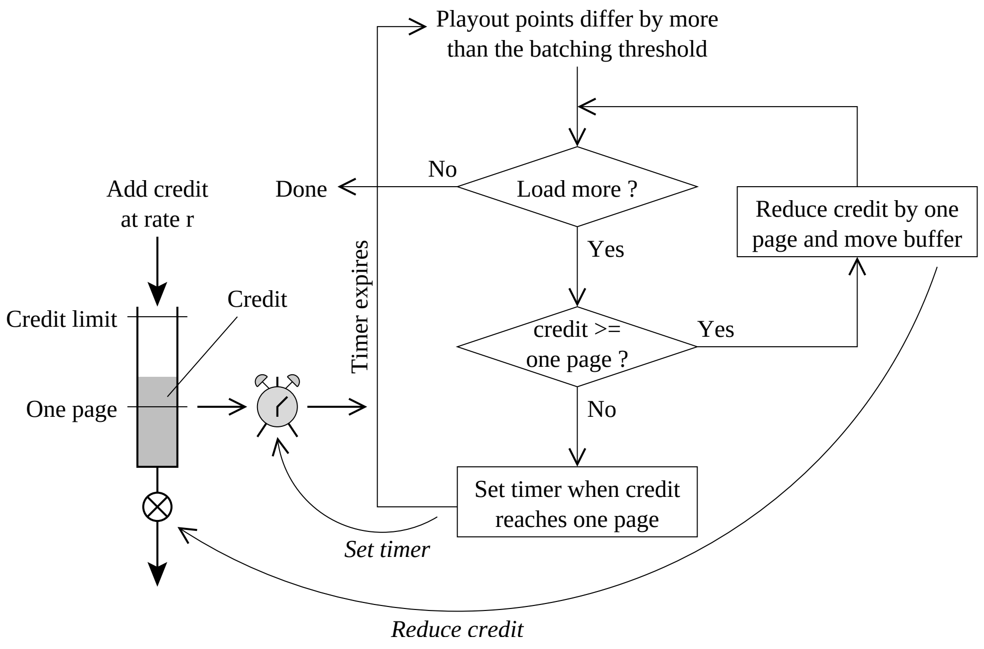
If there is enough credit, the playout buffer is moved immediately. Otherwise, the scheduler sets a timer to expire when enough credit will have accumulated. When the playout buffer is moved, the first page in it is dropped,[7] the remaining pages are shifted by one position, and the new page is requested. If a request for this page is already in progress, the scheduler informs the elevator that the request should now be processed with real-time priority (see section 3.5).

If the playout buffer has to be moved by several pages, the procedure is repeated.

Playout points

The playout buffer "moves" over the file by removing pages at its left-hand side, and loading new pages at its right-hand side. As shown in figure 10, the movement is controlled by two playout points: the application playout point indicates the location after which the application

7. The playout buffer, which is organized as a ring buffer, only contains pointers to the page structures, so dropping a page means to release the reference.



Playout points differ by more than the batching threshold
Load more ?
No
Done
Yes
Reduce credit by one page and move buffer
credit >= one page ?
Yes
No
Set timer when credit reaches one page
Timer expires
Add credit at rate r
Credit
Credit limit
One page
Set timer
Reduce credit

Figure 11: Playout buffer movement is limited by a credit that accumulates at the rate requested by the application, and which is spent when the playout buffer advances through the file.

will access the file content. The application can move this playout point at any time and to any position.

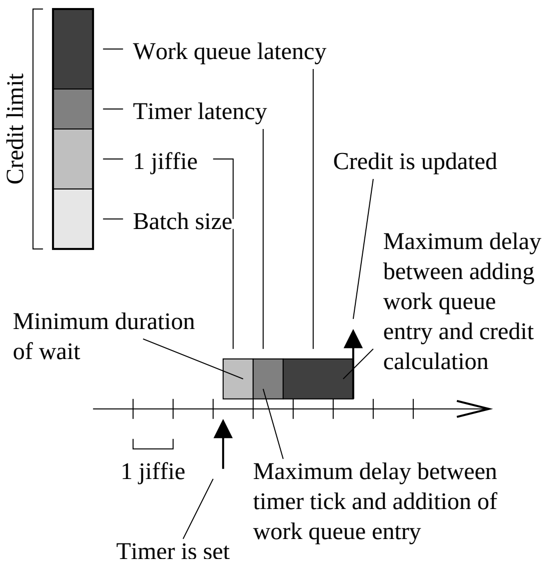
The second playout point is maintained by the scheduler in the kernel. It follows the application playout point, but always moves forward, and its average speed does not exceed the requested rate.

If both playout points are on the same memory page, the playout buffer stops moving. If the application moves its playout point outside the playout buffer, e.g. because it does a "fast forward" or because it exceeds the read rate, it loses the real-time guarantees, the kernel playout point jumps directly to the location of the application playout point, and the buffer is refilled at a best-effort priority.

Rate control

The average rate of movement is limited to the rate the application requested: a movement credit is accumulated at that rate, and whenever the playout buffer moves, some of this credit is spent. If the credit is too small, the scheduler sets a timer that will expire when the credit is sufficient to move the playout buffer by at least one page. This process is illustrated in figure 11.

The credit serves two purposes: (1) it allows the scheduler to handle time with more accuracy than solely relying on timer expiration would, and (2) it lets the scheduler compensate for delays between deciding to initiate a series of read requests, and the moment when these requests are actually issued.



Credit limit
Work queue latency
Timer latency
1 jiffie
Batch size
Credit is updated
Maximum delay between adding work queue entry and credit calculation
Minimum duration of wait
1 jiffie
Maximum delay between timer tick and addition of work queue entry
Timer is set

Figure 12: The limit keeps the scheduler from accumulating excessive credit, while allowing it to compensate for the delays occurring when scheduling IO operations.

Since credit is accumulated whenever the playout window is stopped, an arbitrary amount of credit might be accumulated, and could then be used to issue a large number of real-time read operations, which would disturb overall system performance, and make it impossible to usefully predict delays. Therefore, the maximum credit must be limited to a reasonable value.

As shown in figure 12, the credit consists of the following parts:

- The credit required before the playout window moves at all. This is simply the batch size, as described below.
- The accuracy of timers. Since the scheduler always rounds up to the next higher entire jiffie, the maximum inaccuracy is the timer resolution, i.e. one jiffie.
- The delays between nominal timer expiration and the moment when credit is used to initiate IO operations. These delays depend on how the scheduler is implemented, and include – in the current design – the time to act on timer expiration, plus the time between enqueuing a work queue entry and the time it gets executes.

Applications may use a similar algorithm to time their own playout point movements.

Dimensioning the playout buffer

The playout buffer maintained by the scheduler absorbs all deviations from an ideal constant-rate flow. In section 2.6, we have already discussed the buffering requirements determined by the application. They are shown in the upper part of figure 13.[8]

The lower part of figure 13 shows the additional buffering needed to compensate for effects caused by elements under the responsibility of the operating system:

- A considerable amount of time may pass between the moment, when the scheduler should fetch a new page, and the time when the request is actually enqueued. In the current implementation, this includes the time until the scheduler actually becomes aware that it should issue a new request, and, since we process playout buffer movements through a work queue, the time it takes until the work queue item is processed.
- The time the request spends in the elevator (waiting for other requests to complete[9]), and then the time it takes for the disk to process the request.

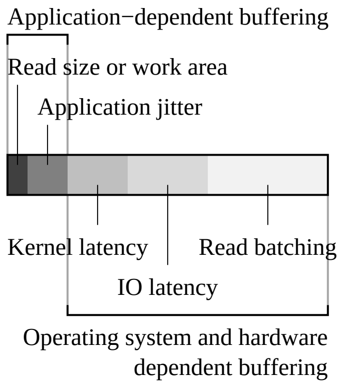

Application−dependent buffering
Read size or work area
Application jitter
Kernel latency
Read batching
IO latency
Operating system and hardware dependent buffering

Figure 13: The playout buffer of the scheduler provides for buffering needs resulting from application properties and from latencies caused by the operating system and the hardware.

- Any *batching* performed by the scheduler. Batching means that the scheduler does not load each page immediately when it can, but waits until a certain minimum number of pages needs to be loaded, and then requests them all at once. This way, high-priority requests interfere less frequently with lower priority requests, allowing the latter to benefit more from the request ordering done by the elevator.

When requesting a real-time service, the application only specifies its own buffering requirements. The kernel and the ABISS daemon then increase the buffer size to include the additional buffering needed for kernel and hardware.

3.3 API example

Communication between the application and the ABISS scheduler is done with an ioctl. To request an ABISS service on a file, the application must first open the file, then fill in a message structure with a scheduler-specific parameter block, and finally issue the ioctl, as shown in the following skeleton code:

```
static struct abiss_attach_msg msg;
static struct abiss_sched_test_prm prm;

fd = open("name", O_RDONLY);
```

8. For simplicity, we subsume everything related to non-ideal behaviour under "jitter".

9. This includes all other requests at real-time priority earlier in the queue, plus a best-effort request that may be executing at that time. On devices with a slow transfer rate, the maximum size limit for requests may have to be lowered to prevent them from taking too much time.

```
msg.header.type = abiss_attach;
msg.sched_prm = &prm;
...
if (ioctl(fd, ABISS_IOCTL, &msg) < 0)
    /* handle error */;
```

Typically, the only other change required is to update the playout point after reading from the file. Again, a message structure is used for this purpose, as shown in the following code fragment:

```
static struct abiss_position_msg msg;

got = read(fd, buffer, BUFFER_SIZE);
msg.header.type = abiss_position;
msg.pos = 0;
msg.whence = SEEK_CUR;
ioctl(fd, ABISS_IOCTL, &msg);
```

In this example, the playout point it set to the current file position (called the *file offset* in [2]).

There is a also a library providing slightly easier to use wrapper functions for these operations.

3.4 Priorities

The mainstream Linux kernel currently has no provision for specifying the priority of IO requests. We build partly upon a mechanism for a process-based "IO priority" that was proposed by Jens Axboe a while ago [4], and that is poised to be added to the Linux kernel [5].

Unfortunately, it is not possible to pass priorities directly along with the operations that eventually lead to an IO request. Instead, each process or thread has its own "IO priority" which applies to all disk IO operations executing under this process.

ABISS uses real-time priorities only indirectly, in the work queue thread that prefetches pages. We therefore set the IO priority of that kernel thread to the real-time priority, before it starts prefetching pages, and return it to its previous value when done.

Independently from this, processes can set their own IO priority for any operations not involving ABISS.

3.5 Upgrades

When the scheduler tries to prefetch a page that has already been requested, that request may be at a lower priority and may have to be *upgraded.* This can happen if the application has slightly exceeded its read rate, and attempted to access a page beyond the playout buffer a moment before the playout buffer shifts to cover this page, but also if a different application is reading the file at non-real-time priority.

When upgrading a page, the scheduler tells the elevator to look for the request that includes the page, and to move that request to the corresponding higher priority queue. If the request covers pages with different priorities, the highest priority is used.[10]

3.6 Elevator

The ABISS elevator provides the infrastructure for enforcing IO priorities. Besides implementing priorities, it differs in a few other regards from the regular elevators in the Linux kernel, i.e. deadline, anticipatory, and CFQ (Complete Fairness Queuing):

- It reserves space in the request queue for high-priority requests, so that they do not have to compete for request queue slots on equal terms with lower-priority requests.
- Barriers are handled in a way that is relatively unobtrusive for read operations. As a welcome side effect, request ordering semantics also become more intuitive.[11]
- Since we expect that the ABISS elevator may be used with comparably large request queue sizes, it serializes requests before a barrier at $O(p)$ instead of the $O(n)$ required by the regular elevators, for p priorities and n pending requests.

10. This may lead to upgrading a possibly large number of pages that should not (or not yet) be retrieved at real-time priority, and might cause the real-time priority queue to grow such that other deadlines will be missed. Since this extension does not cause long-range seeks, the impact should normally be low, and the ABISS daemon can compensate by slightly enlarging the playout buffers of files open for real-time reading. On slow disks, one may also have to adjust the maximum request size.

Doing the opposite, i.e. keeping the request at a lower priority until all pages have been upgraded, yields even less predictable effects, and is likely to cause more pronounced deadline slips.

The correct solution would be to split this request into a high and one or two low priority parts. Unfortunately, this conflicts somewhat with the design of the elevator subsystem in the Linux kernel.

Combined requests, as described in the following section, no longer take part in the priority scheme, and are never upgraded.

11. There is some controversy over whether making ordering semantics in general more predictable is truly desirable, or whether this is a misguided attempt at implementing semantics that cannot be guaranteed in other scenarios anyway. Fortunately, the ABISS elevator can easily be changed to implement either behaviour.

The ABISS elevator is mainly meant for exploring performance and implementation issues related to prioritized IO, and it currently does not aim to offer balanced performance for complex loads, like the anticipatory or CFQ elevators do.

In the future, we will try to merge the functionality of the ABISS elevator that is not specific to ABISS (that is, almost everything) into the CFQ elevator.

Enqueuing requests

Figure 14 shows the data structures in the elevator. It is divided into two *areas*, one for reads, the other for writes and requests with special ordering requirements. In each area, there are eight *priority queues* – one for each priority.

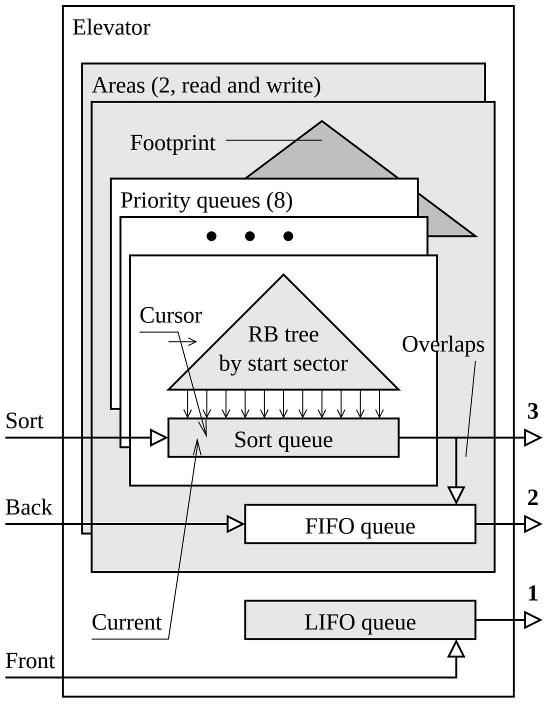


Figure 14: Data structures in the ABISS elevator.

Regular requests are added to the *sort queue*, a linear list, which is ordered by the start sectors of the requests in it. The ordering is accomplished through a red-black tree. Requests which may not be reordered with respect to other requests are placed in either a *FIFO queue*,[12] or the *LIFO queue*. Requests that are requeued by a block device are added at the head of the respective FIFO queue.

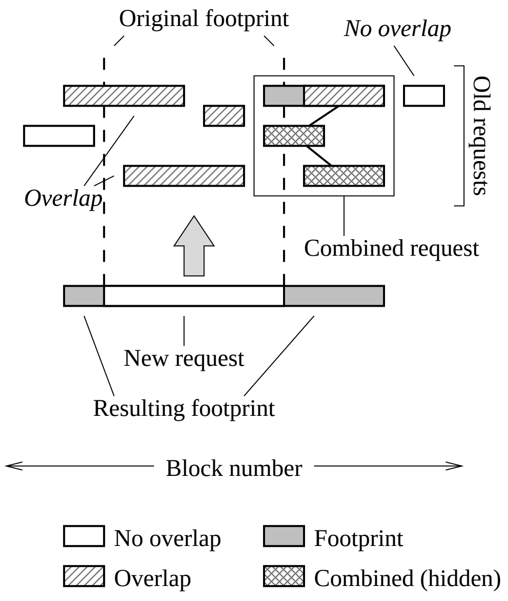


Figure 15: The "footprint" of a request is the range of sectors covered by all the overlapping requests it is combined with.

Retrieving requests

When looking for the next request, the LIFO queue has priority over anything else. If the LIFO queue is empty, the FIFO queue in the currently active area is searched.[13] Finally, if also the FIFO queue was empty, the priority queues of the currently active area are searched, from highest to lowest.

When retrieving the next request from a sort queue, the request pointed to by the so-called *cursor* is taken. The cursor moves from requests beginning at low sectors towards those at higher sectors. When reaching the last request in the sort queue, it wraps back to the beginning. This way, each priority implements a single-sweep elevator.

The elevator alternates between reading and writing. Each phase is given a certain amount of time.[14] A phase ends if either that amount of

12. Usually, all requests going to a FIFO queue go to the write area. The only exception to this are fully overlapped read requests, which, after being retrieved from the sort queue, go to the FIFO queue of the read area instead (see below).

13. Except if the request that was returned last has not yet been removed from the queue, and no other request was enqueued in the LIFO queue. This mechanism is there, because an elevator must consistently return the same "current" request until that request is explicitly removed.

14. At the time of writing, we use 2 seconds for the read phase, and 30 ms for the write phase.

time has passed, or if there are no more requests of the corresponding type. The phase change is postponed if the other area contains no requests.

Barrier semantics

Barriers and overlapping requests require special treatment. In the regular elevators on Linux, barriers separate all requests before and after them, and the elevators give no guarantees with respect to the ordering of overlapping requests. While barriers are rarely used, delaying new requests until all pending requests have been processed may cause significant delays also for higher-priority requests.

Fortunately, it makes no difference if we reorder read requests even across barriers, as long as they are not moved beyond write requests accessing the same disk sectors. The ABISS elevator therefore honors barriers only for write requests, and ensures that read requests never cross write requests which whom they overlap. For simplicity, we always avoid reordering write requests that overlap with other write requests. This has the added benefit that data read and written with the ABISS elevator is exactly the same as if a simple FIFO was used.

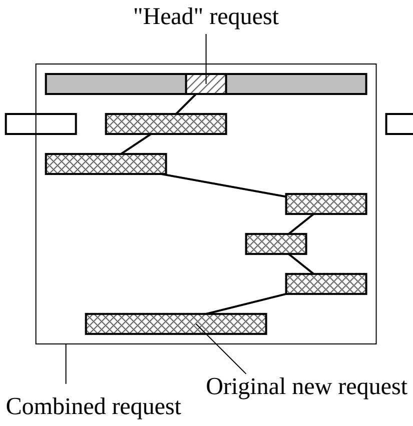


Figure 16: The resulting request combines all requests overlapping with the original new request.

Overlapping requests

Overlapping requests can only occur in combination with operations that bypass the page cache, i.e. direct reads and writes by file system drivers, or files opened with `O_DIRECT`. Overlapping parts of transfers going through the page cache only cause disk IO once, and are resolved within the cache with FIFO semantics.

New write requests are checked for overlaps with existing read and write requests, while new read requests are only checked against existing write requests. If overlaps are found, all overlapping requests are combined in a list such that the existing requests appear in an arbitrary order, followed by the new request. Only the first request of this list is visible in the sort queue and all the trees. In order to account for the requests hidden behind this first request, we introduce the concept of a request's *footprint*, which is the range of sectors used when looking for overlaps. When combining requests, the footprint of the first request is increased accordingly. This procedure is illustrated in figure 15.[15]

If any of the overlapping requests are already combined requests, all its components are added as individual requests (i.e. there are no combined requests nested inside other combined requests), but they retain their relative order.

Figure 16 shows a possible result of this operation. Note that the new request that caused the overlaps is last in the combined request.

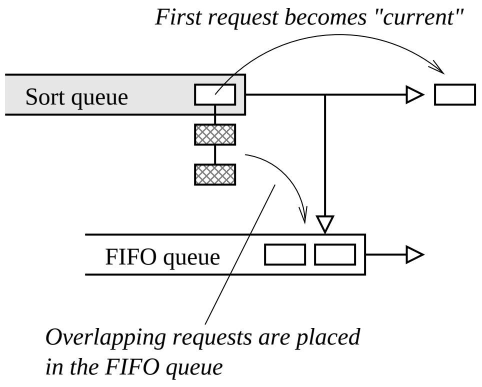

First request becomes "current"
Sort queue
FIFO queue
Overlapping requests are placed in the FIFO queue

Figure 17: Combined requests are split when retrieving them.

Overlapping requests are detected by looking up the range of sectors they cover in a radix priority search tree maintained at each area. This is the dark tree looming in the background on

15. We are exercising a little artistic freedom here: the configuration in this example could not occur in real life: Overlapping requests like the two on the left would always be combined if they were in a write area, so this must be a read area. However, in a read area, there would be no reason to combine the middle request of the combined request.

Requests that cover an identical range of sectors, like the first and the last request in the combined request, would be combined even in a read area, because the search tree we use cannot accommodate identical entries. Therefore, in that exceptional case, the combined request stays in the read area.

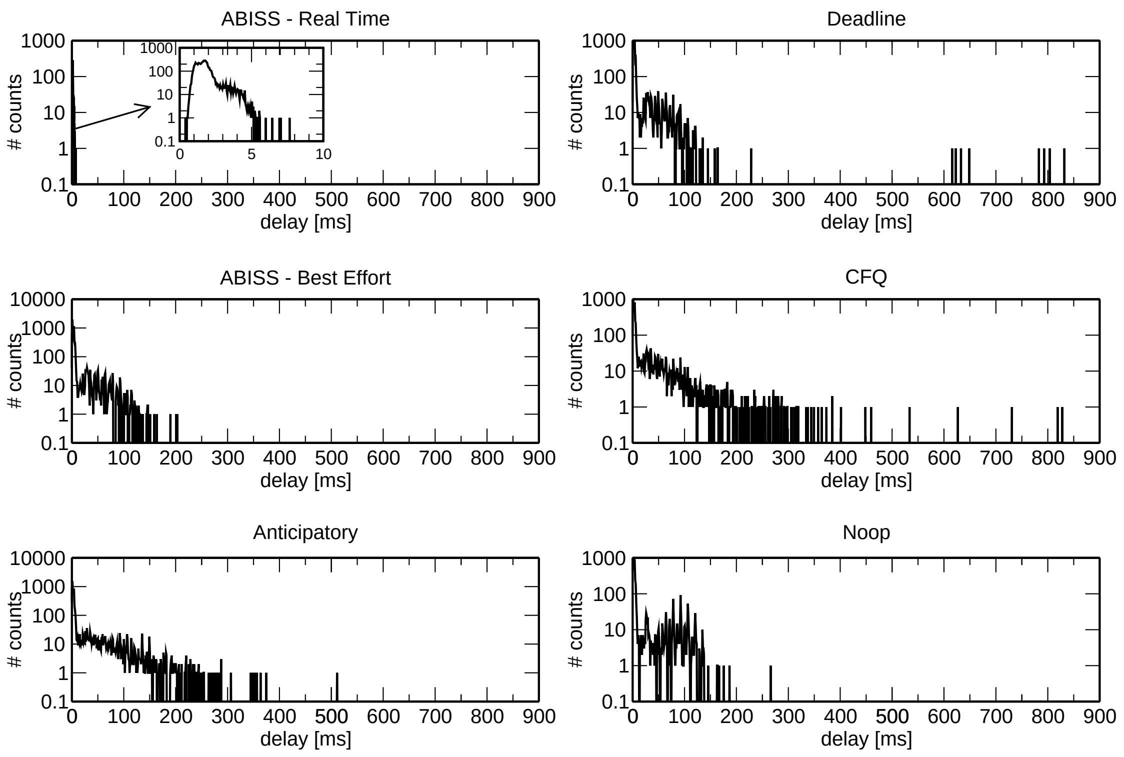


Figure 18: Histogram of the time between issuing a read() call and obtaining the data for the ABISS elevator in RT and BE mode, and for the other Linux elevators. The measurement was done with four real-time streams reading at 1 MB/s and a continuous best-effort read of a large file in the background. In the case of the real-time ABISS elevator the maximum delay is around 8 ms. The batch size for this measurement was 20 pages.

figure 14. After combining requests, the result is enqueued in the write area, since the combined request contains at least one write.

When retrieving a combined request, the "head" becomes the next (current) request, while the rest is separated and placed in the FIFO queue. This is shown in figure 17.

The structure of the ABISS elevator also allows barriers to be implemented very efficiently: since they only affect the write area, it is sufficient to add the contents of all sort queues to the FIFO queue, which is an $O(1)$ operation for each queue. Because this empties all trees in this area, their elements do not need to be removed individually, but the trees can simply be initialized to their empty state.

3.7 Known problems

A general and difficult to solve problem are delays that may appear anywhere along the code paths involved in processing IO requests. In particular, memory management can trigger scans for free pages, with a significant run time. Our current work-around is to apply generous buffering.

Another (minor) problem of best-effort priorities is that requests cannot be upgraded, because ABISS is not participating in request generation.

Real-time reads currently begin as soon as the batch size is reached. This means that the time intervals for best-effort afforded by these batches get fragmented into smaller intervals if there is more than one real-time reader. This defeats the purpose of read batching.

Last but not least, ABISS presently provides no guarantees for writes. There is experimental infrastructure in place that allows ABISS to control where free space is allocated, and that also helps to eliminate meta-data accesses (which, if they are reads, may block).

4 Measurements

In this section we will compare the performance of ABISS to that of the other Linux elevators: Anticipatory (the default in the Linux 2.6.7 ker-

Elevator	Foreground readers	Background reader [MB/s]		Playout buffer
		with 1 foreground	with 4 foreground	
ABISS	RT, 10 page batch	7.7	0.27	564 kB
	RT, 20 page batch	8.0	2.5	564 kB
	RT, 40 page batch	8.7	4.0	564 kB
	RT, 80 page batch	9.4	5.8	564 kB
	RT, 160 page batch	9.5	6.6	1064 kB
	BE	7.7	1.5	—
Anticipatory		7.8	2.7	—
Deadline		7.9	1.8	—
CFQ		7.9	1.8	—
Noop		7.9	2.0	—

Table 1: Data rate obtained by a "background" best-effort reader against one and four concurrent "foreground" best-effort or real-time readers.

nel), Completely Fair Queuing (CFQ), Deadline and Noop. The measurements were done on a system with 128 MB of memory, and a Transmeta Crusoe TM5800 [6] CPU, running at 800 MHz. Two hard disks were connected to the system: the primary `/dev/hda`, containing the boot and system partitions was a 2.5 inch 60 GB hard disk, the secondary hard disk `/dev/hdc` was a 2.5 inch 20 GB 4200 rpm Hitachi Travelstar hard disk [7], with only one partition.

Two tests were performed in which one or four simultaneous streams, reading different 105 MB files on the secondary hard disk, were started. The streams were started with the `rdrt` tool, part of the ABISS distribution, which reads a certain file at a predefined data rate. The data were read in blocks of 64 kB, at a rate of 1 MB/s. The playout buffer size was set to 564 kB. The `rdrt` tool allows logging the delay between issuing a read command and the arrival of the data in the application. In parallel to these real-time streams, which we shall call our *foreground* readers, a *background* best-effort read stream on a fifth large file was started by running a program that continuously reads a 175 MB file in chunks of 128 kB, using fread().

Several measurements were done: one in which the ABISS real-time service was used, one in which the ABISS elevator was used, but only for best-effort (default priorities for both the four foreground streams as well as for the background reader). In the other measurements the other Linux elevators were used. For the measurements with the ABISS real-time service several batch sizes were tried: 10, 20, 40, 80 and 160 pages (one page is 4 kB).

The results of the measurements with four real-time streams are shown as a histogram in figure 18. For a certain delay time on the x axis the number of times this delay occurred is on the y axis. The results of all four real-time streams were summed.

The benefits of using real-time ABISS can be clearly seen; all delays are smaller than 8 ms which means that applications can do with a very small buffer. In the case of best-effort traffic the delays that occur when reading can be up to a second, which implies for this case that the applications will need a buffer of at least 1 MB (and will also incur a latency to user input of at least 1 second). Figure 18 shows the result with a batch size of 20 pages. In other measurements sometimes one or two reads were seen with a delay up to 25 ms. These have not been explained yet.

It is also interesting to see what data rate the best-effort reader could obtain. These rates are listed in table 1. It can clearly be seen that the rate is strongly dependent on the batch size. However, already with a 40 page batch size, the impact of the real-time readers on the background reader is lower than if using the same number of best-effort readers. As expected, the other elevators perform better than the ABISS elevator under a pure best-effort load, but still show degradation well beyond that experienced with properly tuned ABISS real-time readers.

5 Conclusions and future work

In this paper we present our work done on real-time file I/O on hard disks. The ABISS framework allows for different services that can be implemented in modules that can be changed at run time. In this paper we have shown results with a performance scheduler and compared to results obtained with the standard best-effort way of scheduling I/O. By using ABISS the system is able to guarantee a certain bandwidth to an application, without the need for large memory buffers on the application side.

Writing with real-time guarantees is not yet supported. The main challenge in writing large files is to prevent fragmentation of the files. For instance, in the current file systems writing multiple files simultaneously will result in relatively small, interleaved areas on the hard disk. To circumvent this, work has to be done on the allocator, the entity that assigns disk blocks to files that are written.

Future work will also include different ABISS services. For instance, for portable systems power is very important. An ABISS scheduler might allow for power management thus extending the battery life of a portable device.

Furthermore, since there is general interest in functionality to let also the mainstream kernel differentiate IO services, we will merge mechanisms that have been successfully used in ABISS, and submit them for inclusion into the 2.6 or 2.7 kernel.

References

[1] Li, Hong; Cumpson, Stephen R.; Korst, Jan; Jochemsen, Robert; Lambert, Niek. *A Scalable HDD Video Recording Solution Using A Real-time File System.* IEEE Transactions on Consumer Electronics, Vol. 49, No. 3, 663–669, 2003.

[2] *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition*, The IEEE and The Open Group, 2003. `http://www.opengroup.org/onlinepubs/007904975/`

[3] Piggin, Nick. *Runtime selectable IO schedulers.* `http://www.kerneltrap.org/~npiggin/elevator/`

[4] Axboe, Jens. *[PATCH] cfq + io priorities.* Posted on the linux-kernel mailing list, November 8, 2003. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0311.1/0019.html`

[5] Axboe, Jens. *Linux Block IO—present and future.* Proceedings of the Linux Symposium, vol. 1, pp. 51–61, Ottawa, July 2004. `http://www.finux.org/Reprints/Reprint-Axboe-OLS2004.pdf`

[6] The Transmeta Corporation `http://www.transmeta.com/crusoe/crusoe_tm5800_tm5500.html`

[7] Hitachi Global Storage Technologies, disk model number IC25N020ATMR04, `http://www.hitachigst.com/hdd/support/80gn/80gn.htm`