Active Block I/O Scheduling System (ABISS)

Giel de Nijs & Benno van den Brink
Philips Research
{giel.de.nijs,benno.van.den.brink}@philips.com

Werner Almesberger
werner@almesberger.net

Abstract

The Active Block I/O Scheduling System (ABISS) is an extension of the storage subsystem of Linux. It is designed to provide guaranteed reading and writing bit rates to applications, with minimal overhead and low latency.

In this paper, the various components of ABISS as well as their actual implementation are described. This includes work on the Linux elevator and support for delayed allocation.

In a set of experimental runs with real-life data we have measured great improvements of the real-time response of read and write operations under heavy system load.

1 Introduction

As storage space is getting cheaper, the use of hard disk drives in home or mobile consumer devices is becoming more and more mainstream. As this class of devices like HDD video recorders, media centers and personal audio and video players were originally intended to be used by one person at a time (or by multiple persons, but watching the same content), performance of the hard disk drives was not a real issue. Adding more video sources to such a device (more tuners, for instance), however, will strain the storage subsystem by demanding the recording of multiple streams simultaneously. As these devices are being enabled with connectivity options and become interconnected through home networks or personal area networks, a device should also be able to serve a number of audio or video streams to multiple clients. For example, a media center should be able to provide a number of so-called media extenders or renderers throughout the house with recorded content. Putting aside high bit rate tasks, even simple low-end devices could benefit from a very low latency storage system.

Consumer electronics (CE) equipment has to consist of fairly low-cost hardware and often has to meet a number of other constraints like low power consumption and low-noise operation. Devices serving media content should therefore do this in an efficient way, instead of using performance overkill to provide their soft-real-time services. To be able to accomplish this sharing of resources in an effective way, either the applications have to be aware of each other or the system has to be aware of the applications.

In this paper we will present the results of work done on the storage subsystem of Linux, re-

sulting 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 introduces priority-based best-effort (BE) disk traffic.

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

The kernel part consists mainly of our own *elevator* and the ABISS *scheduler*. The elevator implements I/O priorities to correctly distinguish between real-time guaranteed streams and background best-effort requests. The scheduler is responsible for timely preloading and buffering of data. Furthermore, we have introduced an alternative allocation mechanism to be more effectively able to provide real-time writing guarantees. Apart from these new features, some minor modifications were made to file system drivers to incorporate our framework. ABISS supports the FAT, ext2, and ext3 filesystems.

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 a general overview of the ABISS architecture is given. Section 3 describes the steps involved in reading and explains the solutions incorporated in ABISS to control the involved latencies. The same is done for the writing procedure in Section 4. Performance measurements are presented in Section 5, followed by future work in Section 6 and the conclusions in Section 7.

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

2 Architecture

An application reading or writing data from a hard drive in a streaming way needs timely availability of data to avoid skipping of the playback or recording. Disk reads or writes can introduce long and hard-to-predict delays both from the drive itself as well as from the various operating system layers providing the data to the application. Therefore, conventionally a streaming application introduces a relatively large buffer to bridge these delays. The problem however is that as the delays are theoretically unbounded and can be quite long in practice (especially on a system under heavy load), the application cannot predict how much buffer space will be needed. Worst-case buffering while reading means loading the whole file into memory, while a worst-case write buffer should be large enough to hold all the data which is being written to disk.

2.1 Adaptive buffering

If I/O priorities are introduced and thus the involved delays become more predictable, an adaptive buffering scheme may be a useful approach. The adaptive algorithm can compensate for disk latency, system speed and various other variables. Still, an application will need to know how much competition it will face and what the initial parameters should be. Also, the algorithm would need some way to correctly dimension the buffer to be able to sustain some background activity.

Furthermore, some fairness against lower-priority I/O should be maintained. If any application can raise its priority uncontrolled, best-effort traffic can be completely starved. Too

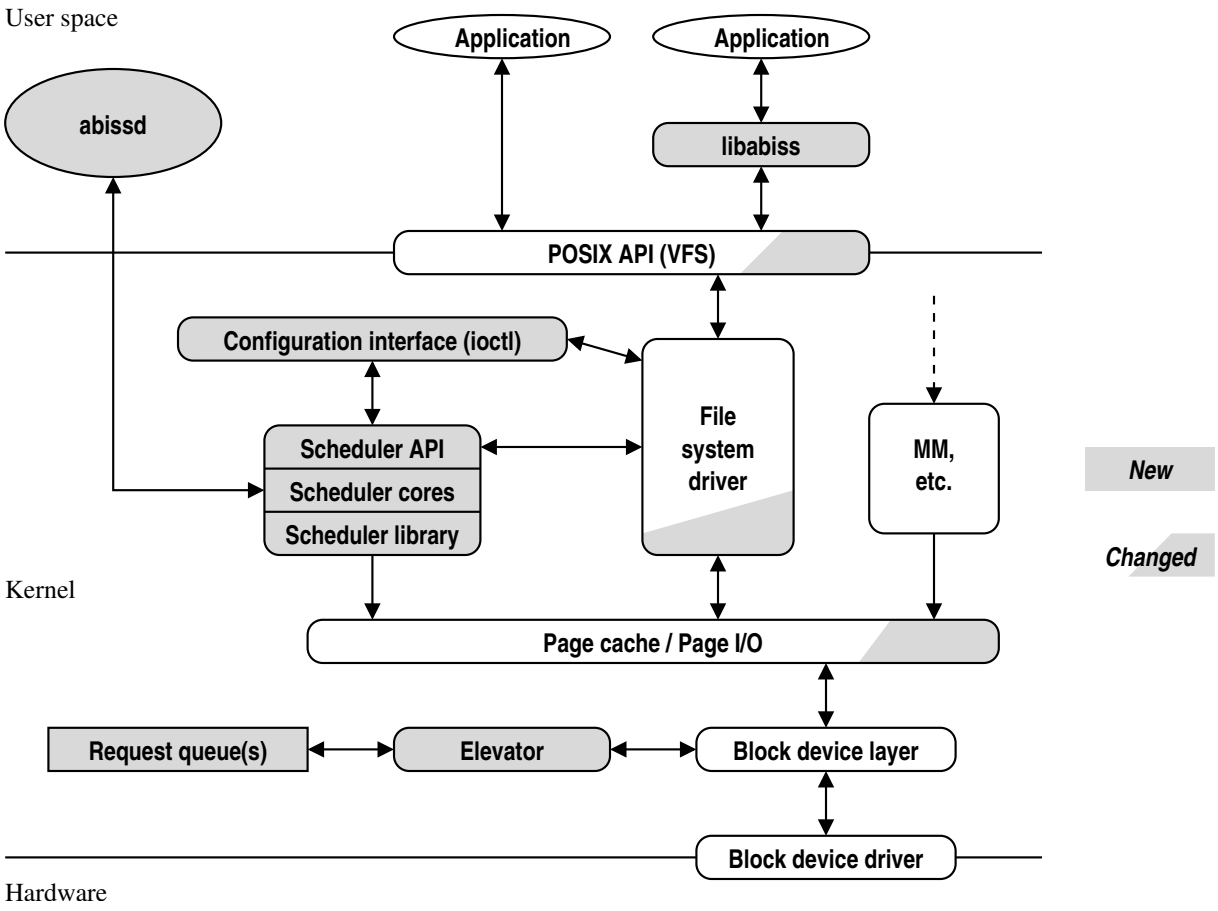

Figure 1: Global ABISS architecture layout.

many applications doing too much I/O at a high priority can also result in unbounded delays for those applications, simply because there are not enough system resources available. Clearly, admission control is needed.

ABISS implements such an adaptive buffering algorithm as a service for streaming applications on a relatively coarse time scale; buffer sizes are determined when the file is opened and may be adapted when the real-time load changes (i.e., when other high-priority files are opened). It makes use of elevated I/O priorities to be able to guarantee bounded access times and a real-time CPU priority to be able to more effectively predict the various operating system related delays. Furthermore, the file system meta-data is cached. All delays are thus predictable in non-degenerate cases and can be caught by a relatively small buffer on system level, outside of the application.

Furthermore, an admission control system is implemented in a user-space daemon to make sure no more commitments are made than the available resources allow. It should be noted that although our daemon offers a framework for extensive admission control, only a very basic system is available at the moment. The architecture of our framework as incorporated in the Linux kernel is shown in Figure 1.

Prior versions of ABISS used very fine-grained administration and measurement instrumentation to have very narrowly defined performance

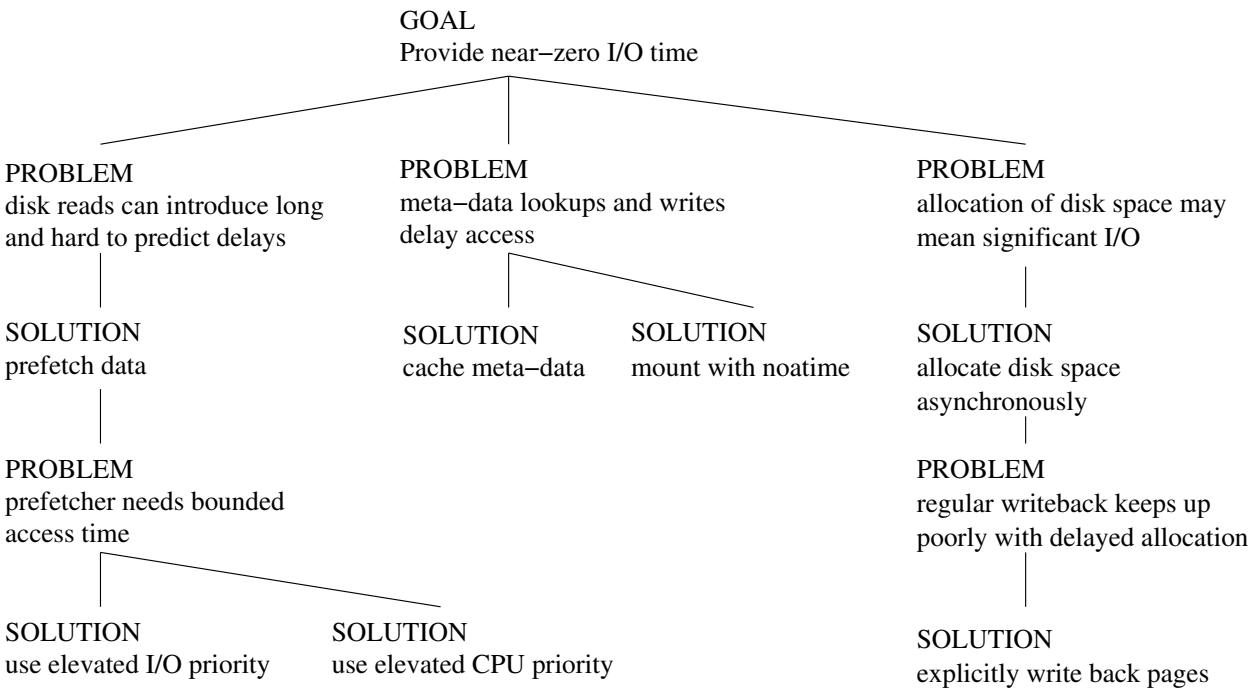

Figure 2: Overview of the solutions incorporated in ABISS.

characteristics. With time, these demands on the underlying layers have gotten "softer." Since we are covering larger parts of the system, leading to influences beyond our full control like the allocation of disk space, we cannot predict the involved delays with such precision as before.

2.2 Service model

When an application requests the services of ABISS (we call such an application an *ABISS user*, or, more specifically, an *ABISS reader* or *writer*), it informs the system about both the bit rate as well as the maximum read or write burst size it is planning to use. A function which opens a file and sets these parameters is available in the ABISS *middleware library*. Given knowledge of the general system responsiveness (I/O latencies, system speed and background load), the buffer can be correctly dimensioned using these variables. This information is also used in the admission control scheme in the daemon which oversees the available system resources.

As the behavior of a streaming application is highly predictable, a fairly simple prefetcher can be used to determine which data should be available in the buffer. The prefetching policy is concentrated in the ABISS scheduler. A separate worker thread performs the actual reading of the data asynchronously, to keep the response time to the application to a minimum.

We use the prefetcher mechanism also when writing, in which case it is not only responsible for the allocating and possibly loading of new pages, but also for coordinating writeback.

To minimize the response time during writing the operations which introduce delays are removed from the calling path of the write operation of the application. This is done by postponing the allocation, to make sure this I/O intensive task is done asynchronously at a moment

the system has time to spare. In our "*delayed allocation*" solution, space for new data in the buffer does not get allocated until the moment of writeback.

An overview of the above solutions is shown graphically in Figure 2. The technical implementations will be elaborated below.

2.3 Formal service definition

The real-time service offered to an application is characterized by a data rate r and a maximum burst read size b. The application sets the *playout point* to mark the location in the file after which it will perform 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 guaranteed to be served from memory.

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 $t_i < t_j$:

$$p_i \leq p_j < p_i + b + r(t_j - t_i)$$
$$p_j \leq a_j < b + \min(p_j, p_i + r(t_j - t_i))$$

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.

3 Reading

When reading a page of file data, the kernel first allocates a free page. Then it determines the location of the corresponding disk blocks, and may create so-called *buffer heads*[1] for them. Next, it submits disk I/O requests for the buffer heads, and waits for these requests to complete. Finally, the data is copied to the application's buffer, the *access time* is updated, and the `read` system call returns. This procedure is illustrated in Figure 3.

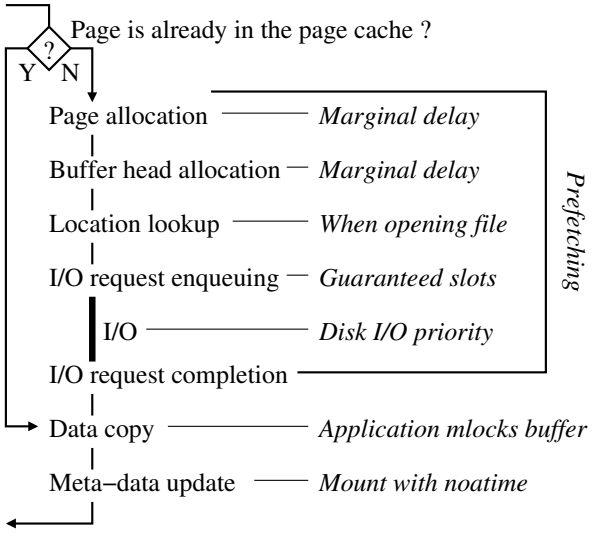

Figure 3: The steps in reading a page, and how ABISS controls their latency.

If trying to read a page that is already present in memory (in the so-called *page cache*), the data becomes available immediately, without any prior I/O. Thus, to avoid waiting for data to be read from disk, we make sure that it is already in the page cache when the application needs it.

3.1 Prefetching

We can accurately predict which data will be read, and can therefore initiate the read process ahead of time. We call this *prefetching*. Pages

[1] A buffer head describes the status and location of a block of the corresponding file system, and is used to communicate I/O requests to the block device layer.

read in advance are placed in a *playout buffer*, illustrated in Figure 4, in which they are kept until the application has read them. After that, pages with old data are evicted from the playout buffer, and new pages with data further into the file are loaded. This can also be thought of as a buffer sliding over the file data.

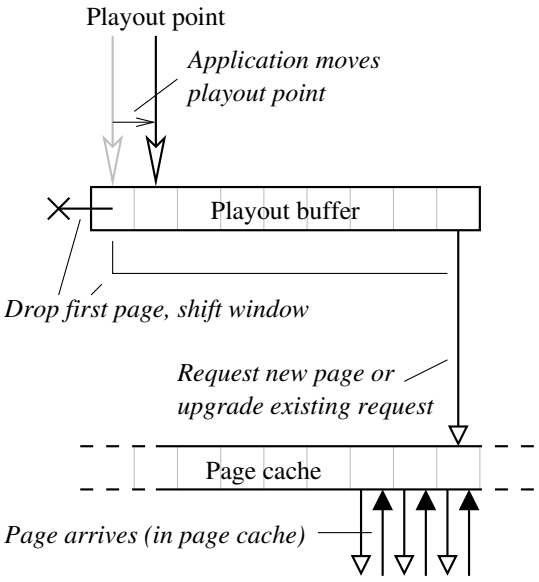

Figure 4: Playout buffer movement is initiated by the application moving its playout point. More than one page may be "in flight" at once.

The playout buffer maintained by ABISS is not a buffer with the actual file data, but an array of pointers to the page structures, which in turn describe the data pages.

Since the maximum rate at which the application will read is known, we can, given knowledge of how long the data retrieval will take, size the playout buffer accordingly, as shown in Figure 5. For this, we consider the space determined by the application, and the buffering needed by the operating system to load data in time. The application requests the total buffer size it needs, which comprises the maximum amount of data it will read at once, and the space needed to compensate for imperfections in its scheduling. To this, buffering is added to cover the maximum time that may pass between initiating retrieval of a page and its arrival, and the batching described in Section 3.4.

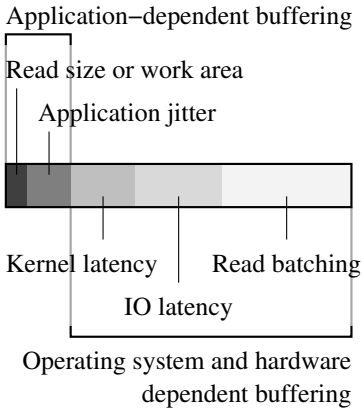

Figure 5: The playout buffer of the scheduler provides for buffering needs resulting from application properties and from latencies caused by the operating system and the hardware.

Prefetching is similar to the *read-ahead* process the kernel performs regularly when sequentially reading files. The main differences are that read-ahead uses heuristics to predict the application behaviour, while applications explicitly tell ABISS how they will read files, and that ABISS keeps a reference to the pages in the playout buffer, so that they cannot be reclaimed before they have actually been used.

Prefetching is done in a separate kernel thread, so the application does not get delayed.

For prefetching to work reliably, and without consuming excessive amounts of memory, data retrieval must be relatively quick, and the worst-case retrieval time should not be much larger than the typical retrieval time. In the following sections, we describe how ABISS accomplishes this.

3.2 Memory allocation

When reading a page from disk, memory allocation happens mainly at three places: (1) when allocating the page itself, (2) when allocating the buffer heads, and (3) when allocating disk I/O request structures.

The first two are regular memory allocation processes, and we assume that they are not sources of delays significantly larger than disk I/O latency.[2]

The number of disk I/O request structures is limited by the maximum size of the request queue of the corresponding device. If the request queue is full, processes wanting to enqueue new requests have to wait until there is room in the queue. Worse yet, once there is room, all processes waiting for it will be handled in FIFO order, irrespective of their CPU priority.

In order to admit high priority I/O requests (see below) instantly to the request queue, the ABISS elevator can be configured to guarantee a certain number of requests for any given priority. Note that this does not affect the actual allocation of the request data structure, but only whether a process has to wait before attempting an allocation.

3.3 Prioritized disk I/O

The key purpose of ABISS is to hide I/O latency from applications. This is accomplished mainly through the use of prefetching. Now, in order to make prefetching work properly, we also have to limit the worst-case duration[3] of

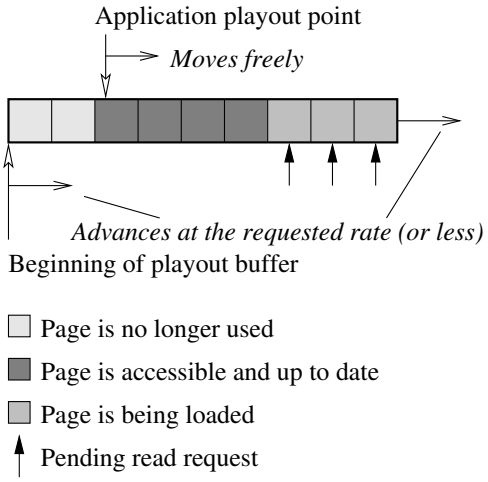

Figure 6: Playout buffer movement is controlled by the kernel, and tracks the position of the playout point, controlled by the application.

I/O requests, independent from what competing applications may do.

ABISS achieves isolation against applications not using ABISS by giving I/O requests issued by the prefetcher thread a higher priority than requests issued by regular applications. The priorities are implemented in the *elevator*:[4] requests with a high priority are served before any requests with a lower priority. We currently use an elevator specifically designed for ABISS. In the future, we plan to migrate to Jens Axboe's more versatile time-sliced CFQ elevator [2].

An interesting problem occurs if a page enters an ABISS playout buffer while being read at a low priority. In order to avoid having to wait until the low priority requests get processed, the prefetcher *upgrades* the priority of the requests associated with the page.

We have described the ABISS elevator in more detail in [3].

[2] In fact, they are much shorter most of the time, except when synchronous memory reclaim is needed.

[3] We ignore degenerate cases, such as hardware errors.

[4] Also called "I/O scheduler." In this paper, we use "elevator" to avoid confusion with the CPU scheduler and the ABISS scheduler.

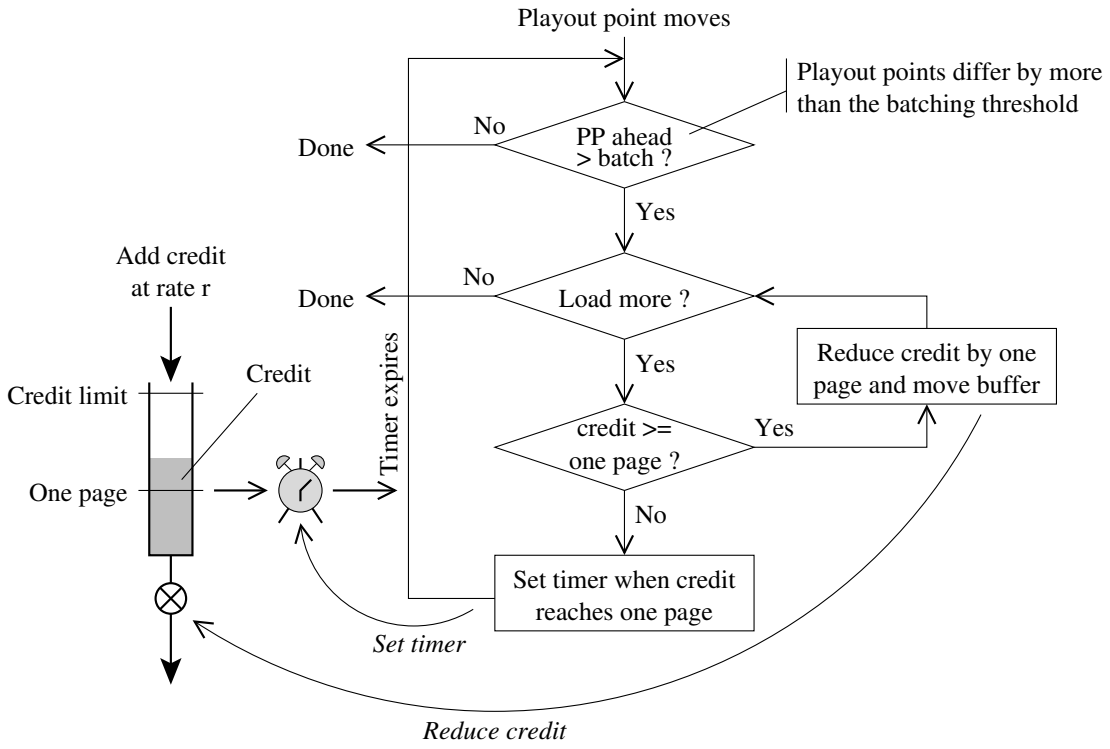

Figure 7: Playout buffer movement is limited by a credit that accumulates at the rate requested by the application, and which is spent when the playout buffer advances through the file.

ABISS users may also compete among each other for I/O. To ensure that there is enough time for requests to complete, the playout buffer must be larger if more ABISS users are admitted. Dynamically resizing of playout buffers is currently not implemented. Instead, the initial playout buffer size can be chosen such that it is sufficiently large for the expected maximum competing load.

3.4 Rate control

Movement of the playout buffer is limited to the rate the application has requested. Application and kernel synchronize through the so-called *playout point*: when the application is done accessing some data, it moves the playout point after this data. This tells the kernel that the playout buffer can be shifted such that its beginning lines up with the playout point again, as shown in Figure 6.

We require explicit updating of the playout point, because, when using `read` and `write`, the file position alone may not give an accurate indication of what parts of the file the application has finished reading. Furthermore, in the case of memory-mapped files, or when using `pread` and `pwrite`, there is no equivalent of the file position anyway.

The ABISS scheduler maintains a *credit* for playout buffer movements. If enough credit is available to align the playout buffer with the playout point, this is done immediately. Otherwise, the playout buffer catches up as far as it can until all credit is consumed, and then advances whenever enough new credit becomes available. This is illustrated in Figure 7.

The credit allows the playout buffer to "catch up" after small distortions. Its accumulation is capped to the batch size described below, plus the maximum latency for timer-driven playout buffer movement, as shown in Figure 8.

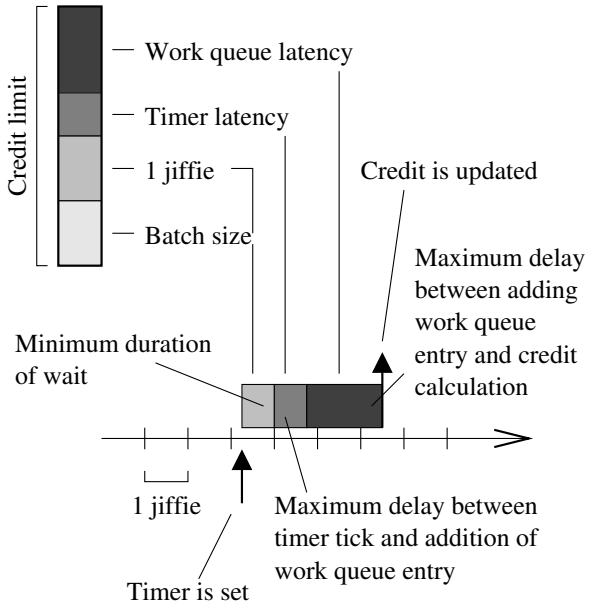

Figure 8: The limit keeps the scheduler from accumulating excessive credit, while allowing it to compensate for the delays occurring when scheduling operations.

If the file was read into the playout buffer one page at a time, and there is also concurrent activity, the disk would spend an inordinate amount of time seeking. Therefore, prefetching only starts when a configurable *batching threshold* is exceeded, as shown in Figure 9. This threshold defaults to ten pages (40 kB).

Furthermore, to avoid interrupting best-effort activity for every single ABISS reader, prefetching is done for all files that are at or near (i.e., half) the batching threshold, as soon as one file reaches that threshold. This is illustrated in Figure 10.

3.5 Wrapping up

Copying the data to user space could consume a significant amount of time if memory for the buffer needs to be allocated or swapped in at that time. ABISS makes no special provisions for this case, because an application can easily avoid it by `mlock`ing this address region into memory.

Finally, the file system may maintain an access time, which is updated after each read operation. Typically, the access time is written back to disk once per second, or less frequently. Updating the access time can introduce particularly large delays if combined with journaling. Since ABISS currently provides no mechanism to hide these delays, file systems used with it should be mounted with the `noatime` option.

4 Writing

When writing a page, the overall procedure is similar to reading, but a little more complicated, as shown in Figure 11: if the page is not already present in the page cache, a new page is allocated. If there is already data for this page in the file, i.e., if the page does not begin beyond the end of file, and does not in its entirety coincide with a hole in the file, the old data is read from disk.

If we are about to write new data, the file system driver looks for free space (which may involve locking and reading file system meta-data), allocates it, and updates the corresponding file system meta-data.

Next, the data is copied from the user space buffer to the page. Finally, the status of the buffer heads and the page is set to "dirty" to indicate that data needs to be written back to disk,

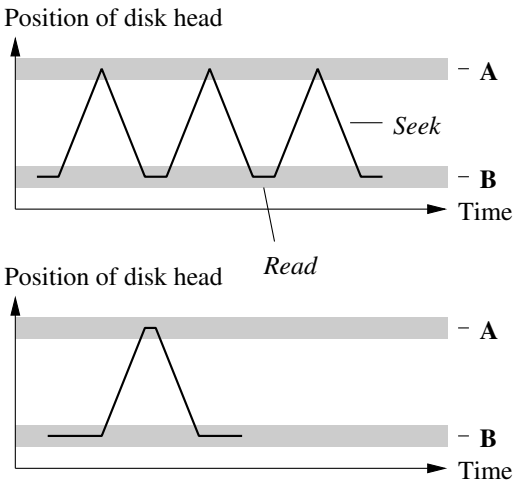

Figure 9: Reading a file (**A**) with ABISS one page at a time (above) would cause many seeks, greatly slowing down any concurrent best-effort reader (**B**). Therefore, we batch reads (below).

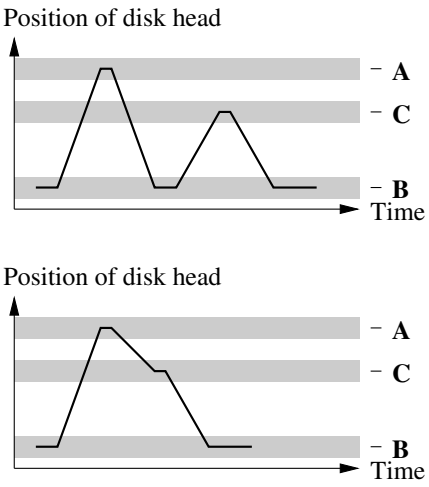

Figure 10: If there are multiple ABISS readers (**A** and **C**), further seeks can be avoided if prefetching is synchronized (below).

and to "up to date" to indicate that the buffers, or even the entire page, are now filled with valid data. Also file meta-data, such as the file size, is updated.

At this point, the data has normally not been written to disk yet. This *writeback* is done asynchronously, when the kernel scans for *dirty* pages to flush.

If using journaling, some of the steps above involve accesses to the journal, which have to complete before the write process can continue.

If overwriting already allocated regions of the file, the steps until after the data has been copied are the same as when reading data, and ABISS applies the same mechanisms for controlling delays.

4.1 Delayed allocation

When writing new data, disk space for it would have to be allocated in the `write` system call. It is not possible to do the allocation at prefetch time, because this would lead to inconsistent file state, e.g., the nominal end-of-file could differ from the one effectively stored on disk.

A solution for this problem is to defer the allocation until after the application has made the `write` system call, and the data has been copied to the page cache. This mechanism is called *delayed allocation*.

For ABISS, we have implemented experimental delayed allocation at the VFS level: when a page is prefetched, the new `PG_delalloc` page flag is set. This flag indicates to other VFS functions that the corresponding on-disk location of the data is not known yet.

Furthermore, `PG_delalloc` indicates to memory management that no attempt should be made to write the page to disk, e.g., during normal writeback or when doing a `sync`. If such a writeback were to happen, the kernel would automatically perform the allocation, and the page would also get locked during this. Since allocation may involve disk I/O, the page may stay locked for a comparably long time, which

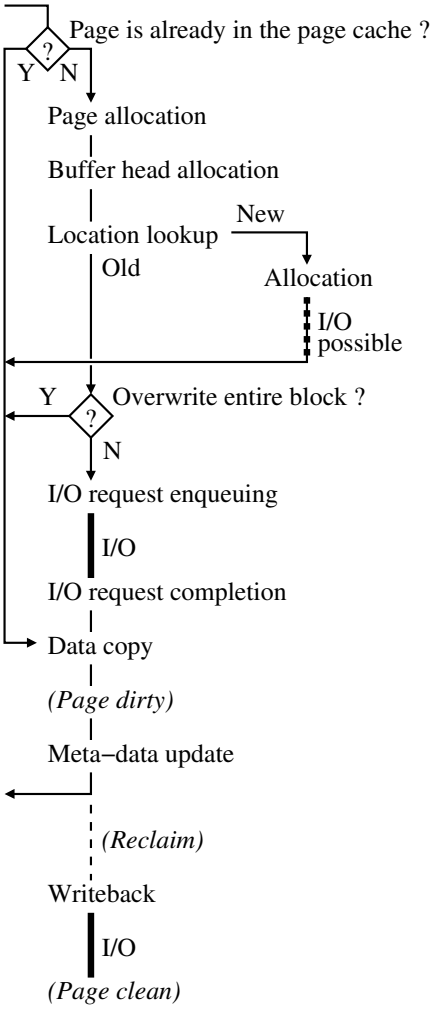

Figure 11: The steps in writing a page (without ABISS).

could block an application using ABISS that is trying to access this page. Therefore, we ensure that the page does not get locked while it is still in any playout buffer.

The code to avoid allocation is mainly in `fs/buffer.c`, in the functions `__block_commit_write` (we set the entire page dirty), `cont_prepare_write` and `block_prepare_write` (do nothing if using delayed allocation), and also in `mpage_writepages` in `fs/mpage.c` (skip pages marked for delayed allocation).

Furthermore, `cont_prepare_write` and `block_prepare_write` may now see pages that have been prefetched, and thus are already up to date, but are not marked for delayed allocation, so these functions must not zero them.

The prefetching is done in `abiss_read_page` in `fs/abiss/sched_lib.c`, and writeback in `abiss_put_page`, using `write_one_page`.

Support for delayed allocation in ABISS currently works with FAT, ext2, and ext3 in `data=writeback` mode.

4.2 Writeback

ABISS keeps track of how many playout buffers share each page, and only clears `PG_delalloc` when the last reference is gone. At that time, the page is explicitly written back by the prefetcher. This also implies allocating disk space for the page.

In order to obtain a predictable upper bound for the duration of this operation, the prefetcher uses high disk I/O priority.

We have tried to leave final writeback to the regular memory scan and writeback process of the kernel, but could not obtain satisfactory performance, resulting in the system running out of memory. Therefore, writeback is now done explicitly when the page is no longer in any ABISS playout buffer. It would be desirable to avoid this special case, and more work is needed to identify why exactly regular writeback performed poorly.

4.3 Reserving disk space

A severe limitation of our experimental implementation of delayed allocation is that errors,

in particular allocation failures due to lack of disk space or quota, are only detected when a page is written back to disk, which is long after the `write` system call has returned, indicating apparent success.

This could be solved by asking the file system driver to reserve disk space when considering a page for delayed allocation, and using this reservation when making the actual allocation. Such a mechanism would require file system drivers to supply the corresponding functionality, e.g., through a new VFS operation.

There is a set of extensions for the ext3 file system by Alex Tomas [4], which also adds, among other things, delayed allocation, along with reservation. Unfortunately, this implementation is limited to the ext3 file system, and extending it to support the prefetching done by ABISS would require invasive changes.

More recent work on delayed allocation with fewer dependencies on ext3 [4] may be considerably easier to adapt to our needs. However, actively preventing allocation while a page is in any playout buffer, which is a requirement unique to ABISS, may be a controversial addition.

4.4 Meta-data updates

When writing, file meta-data such as the file size and the modification time is also changed, and needs to be written back to disk. When reading, we could just suppress meta-data updates, but this is not an option when writing. Instead, we count on these updates to be performed asynchronously, and therefore not to delay the ABISS user.

This is clearly not an optimal solution, particularly when considering journaling, which implies synchronous updates of on-disk data, and we plan to look into whether meta-data updates can be made fully asynchronous, while still honoring assurances made by journaling.

Figure 12 shows the modified write process when using ABISS, with all read and write operations moved into the prefetcher.

4.5 FAT's contiguous files

Files in a FAT file system are always logically contiguous, i.e., they may not have holes. If adding data beyond the end of file, the in-between space must be filled first. This causes a conflict, if we encounter a page marked for delayed allocation while filling such a gap. If we write this page immediately, we may inflict an unexpected delay upon the ABISS user(s) whose playout buffer contains this page. On the other hand, if we defer writing this page until it has left all playout buffers, we must also block the process that is trying to extend the file, or turn also this write into a delayed allocation.

Since our infrastructure for delayed allocations does not yet work for files accessed without ABISS, and because a page can be held in a playout buffer indefinitely, we chose to simply ignore the delayed allocation flag in this case, and to write the page immediately.

A more subtle consequence of all files being contiguous is that new space can only be allocated in a `write` call, never when writing back memory-mapped data. With delayed allocation this changes, and allocations may now happen during writeback, triggered by activity of the allocation code. As a consequence, the locking in the allocation code of the FAT file system driver has to be changed to become reentrant.[5]

[5] This reorganization is partly completed at the time of writing.

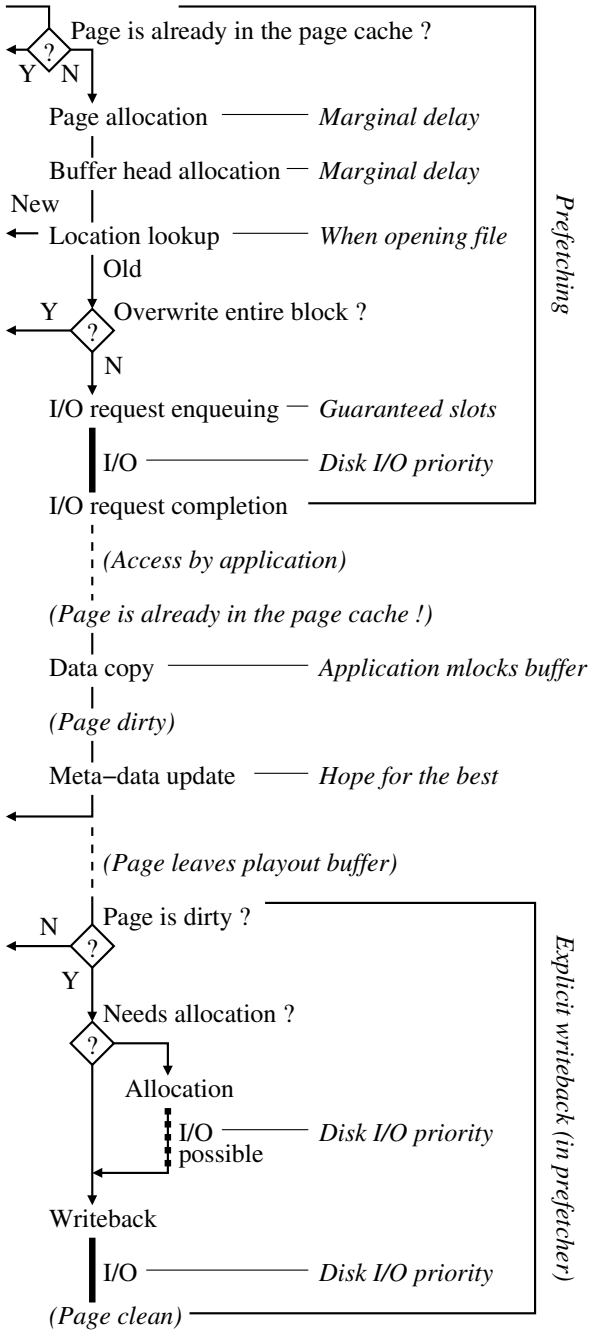

Figure 12: The modified sequence of steps in writing a page using ABISS.

5 Measurements

To be able to assure we have reached our main goal as stated before, near-zero I/O delays, a testing setup was created. The machine running ABISS was deliberately a fairly low-end machine, to assess the results in the light of embedded consumer devices. The data was gathered by `rwrt`, a tool in the ABISS distribution which performs isochronous read or write operations on a file with a certain specified data rate. We have compared the results obtained using ABISS with those obtained using the standard Linux disk I/O. For fair comparison, we used the ABISS elevator on all occasions.

The measurements are performed on a system built around a Transmeta Crusoe TM5800 CPU [5], running at 800 MHz, equipped with 128 MB of main memory of which about 92 MB is available for applications, according to `free`. Two hard drives were connected to the system: the primary drive containing the operating system and applications, and a secondary drive purely for measurement purposes. The drive on which our tests were performed was a 2.5” 4200 RPM Hitachi Travelstar drive.

We have measured the jitter and the latency of reads and writes, the latency of advancing the playout point, the duration of the sleeps of our measurement tool between the I/O calls and the effective distance of the playout point movements. Of these values the jitter is the most interesting one, as it includes both the system call time as well as any effects on time-keeping. Therefore it is a realistic view of what an application can really expect to get. This is further explained in Figure 13. Furthermore, the behaviour of background best-effort readers was analyzed.

Last but not least, we made sure that the streams we read or write are not corrupted in

the process. This was done by adding sequence numbers in the streams, either in prepared streams for reading or on-the-fly while writing.

```
due_time = now;
while (work_to_do) {
    // A (should ideally be due_time)
    read();
    // B
    move_playout();
    // C
    due_time = when next read is due;
    if (due_time < now)
        due_time = now;
    sleep_until(due_time);
}
```

Figure 13: Main loop in rwrt used for reading. Latency is the time from A to B, jitter is B – due_time.[6] Playout point advancement latency is C – B. A similar loop is used for writing. Missed deadlines are forgiven by making sure the next due_time will never be in the past.

5.1 Reading and writing performance

The delays of both the read and write system call with ABISS were measured under heavy system load, to show we are effectively able to guarantee our promised real-time behaviour. Using the `rwrt` tool, we have read or written a stream of 200 MB with a data rate of 1 MB/s, in blocks of 10 kB. The playout buffer size was set to 564 kB for reading and a generous 1 MB for writing, as the latter stressed the system noticeably more. The number of guaranteed real-time requests in the elevator queue was set to 200.

For the tests involving writing, data was written to a new file. The system load was generated by simultaneously running eight greedy best-effort readers or writers[7] during the tests, using separate files with an as high as possible data rate. The background writers were overwriting old data to avoid too many allocations.

5.2 Timeshifting scenario test

To show a realistic scenario for applications mentioned in the introduction of this paper, we have measured the performance of three foreground, real-time writers writing new data, while one foreground real-time reader was reading the data of one of the writers. This is comparable with recording two streams while watching a third one using timeshifting[8]. We have used the same setup as with the previous measurements, i.e., the same bit rate and file sizes.

5.3 Results

The top two plots in Figure 14 show the measured jitter for reading operations. The plots are cumulative proportional, i.e., each point expresses the percentage of requests (on the y-axis) that got executed after a certain amount of time (on the x-axis). For example, a point at (5 ms, 0.1%) on the graph would indicate that 0.1% of all operations took longer than 5 ms. This nicely shows the clustering of the delays; a steep part of the graphs indicates a cluster.

It can be seen that only a small percentage of the requests experience delays significantly longer than average. However, those measurements are the most interesting ones, as we try

[6] We considered using the interval C – due_time instead, but found no visible difference in preparatory tests.

[7] Greedy readers or writers try to read or write as fast as possible, in this case in a best-effort way, using a lower CPU and I/O priority than the ABISS processes.

[8] Timeshifting is essentially recording a stream and playing the same stream a few minutes later. For example, this can be used for pausing while watching a broadcast.

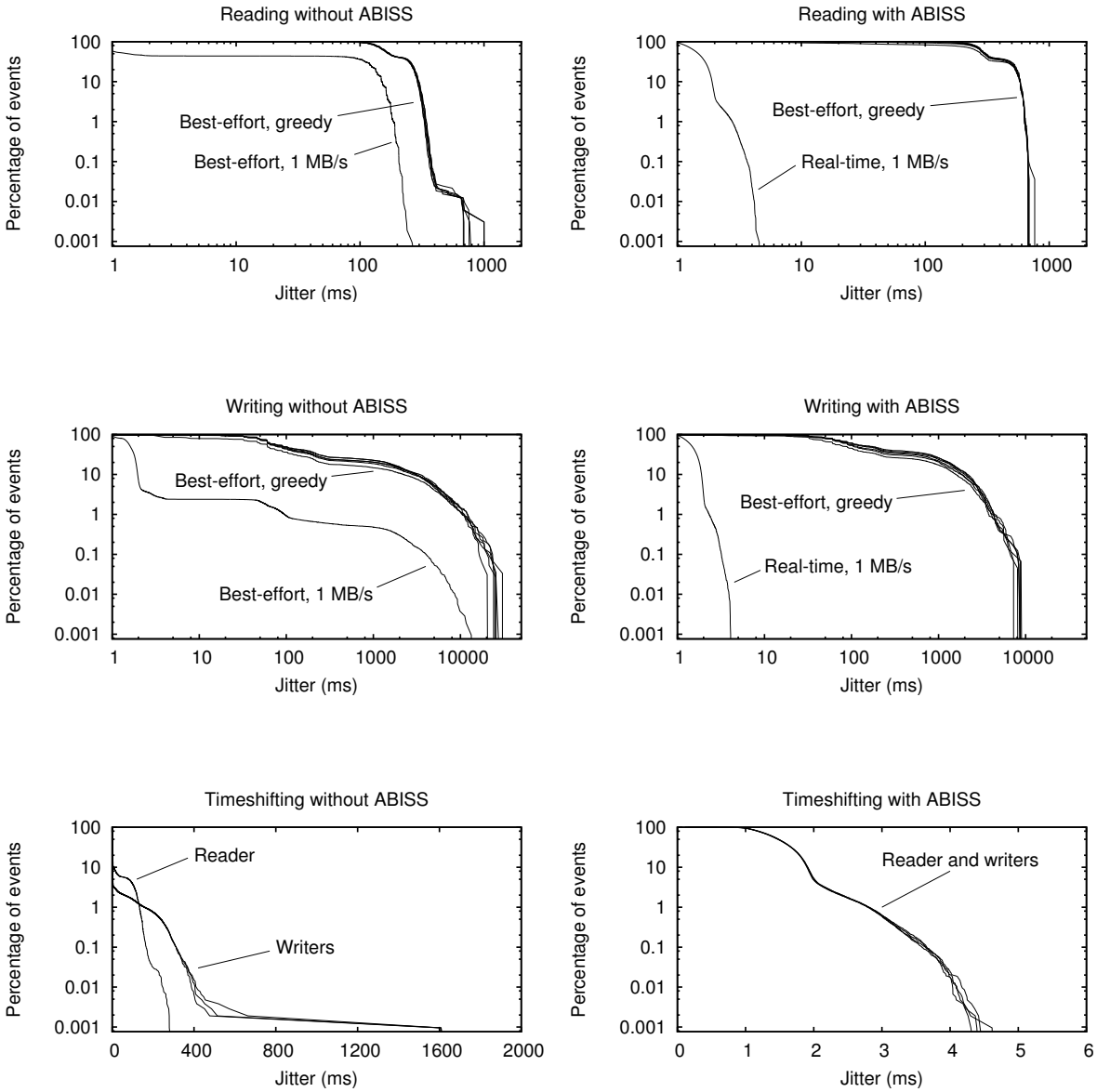

Figure 14: Cumulative proportional plots of the jitter measurements. In all cases the ABISS elevator was used and the measurements were performed on a FAT filesystem.

to bound the experienced delays heuristically. To be able to focus on these delays, the y-axis is logarithmic. As the greedy best-effort readers experience delays of orders of magnitude longer than the real-time delays, the x-axis is logarithmic as well.

Without using the ABISS prefetching mechanism or I/O priorities, all traffic is basically unbounded best-effort. Under the load of the greedy readers, the requested 1 MB/s can definitely not be provided by the system. Although the majority of the requests are served within a few milliseconds, occasional delays of up to a 300 ms were measured. The performance of the greedy readers is even worse: maximum service times of more than a second occurred.

When ABISS is used, we see an enormous decrease of the maximum delay: the reading requests of the 1 MB/s foreground reader now get serviced within less than 5 ms, while the background readers are hardly influenced.

Similar results were observed when using ABISS for writing, as can be concluded from the middle two plots in Figure 14. Using no buffering, prefetching or real-time efforts, but with the ABISS elevator, both the 1 MB/s writer of new data as the greedy background writers experience delays of up to ten seconds. ABISS is able to decrease the service times of the foreground writer to the same level as when it is used for reading: a maximum delay of less than 5 ms, while again the background writers experience little discomfort.

As for the timeshifting scenario with multiple high-priority real-time writers and a ditto reader, the results conform with the above. The results are shown in the last two plots in Figure 14. Without the help of ABISS, especially the writers cannot keep up at all and some request only get served after seconds. Again, using ABISS shortens the delays to less than 5 ms, for both the reader and the writers.

6 Future work

We have briefly experimented with a mechanism based on the NUMA emulator [6], to provide a guaranteed amount of memory to ABISS users. With our changes, we generally observed worse results with than without this mechanism, which suggests that Linux memory management is usually capable to fend for itself, and can maintain sufficient free memory reserves. In periods of extreme memory pressure, this is not true, and additional help may be needed.

When additional ABISS users are admitted or applications close their files, I/O latency changes. In response to this, playout buffers should be adjusted. We currently only provide the basic infrastructure for this, i.e., the ABISS daemon that oversees system-wide resource use, and a set of communication mechanisms to affect schedulers, but we do not implement dynamic playout buffer resizing so far.

Since improvements are constantly being made to the memory management subsystem, it would be good to avoid the explicit writeback described in Section 4.2, and use the regular writeback mechanism instead. We need to identify why attempts to do so have only caused out of memory conditions.

As discussed in Section 4.3, error handling when using delayed allocation is inadequate for most applications. This is due to the lack of a reservation mechanism that can presently be used by ABISS. Possible solutions include either the introduction of reservations at the VFS level, or to try to use file system specific reservation mechanisms, such as the one available for ext3, also with ABISS.

Since delayed allocation seems to be useful in many scenarios, it would be worthwhile to try to implement a general mechanism, that is neither tied to a specific usage pattern (such as the

ABISS prefetcher), nor confined to a single file system. Also, delayed allocation is currently very experimental in ABISS, and some corner cases may be handled improperly.

Last but not least, it would be interesting to explore to what extent the functionality of ABISS could be moved into user space, e.g., by giving regular applications limited access to disk I/O priorities.

7 Conclusion

The ABISS framework is able to provide a number of different services for controlling the way reads and writes are executed. It furthermore allows for a highly controlled latency due to the use of elevated CPU and I/O priorities by using a custom elevator. These properties have enabled us to implement a service providing guaranteed I/O throughput and service times, without making use of an over-dimensioned system. Other strategies might also be implemented using ABISS, e.g., a HDD power management algorithm to extend the battery life of a portable device.

Reading is a more clearly defined operation than writing and the solutions for controlling the latencies involved have matured, yielding good results with FAT, ext2, and ext3. We have identified the problem spots of the writing operation and have implemented partial solutions, including delayed allocation. Although these implementations are currently in a proof-of-concept state, the results are good for both FAT and ext2. The interface complexity of our framework is hidden from the application requesting the service, by introducing a middleware library.

To determine the actual effectiveness and performance of both the framework as well as the implemented scheduler, we have carried out several measurements. The results of the standard Linux I/O system have been compared with the results of using ABISS. Summarizing, using ABISS for reading and writing streams with a maximum bit rate which is known *a priori* leads to heuristically bounded service times in the order of a few milliseconds. Therefore, buffering requirements for the application are greatly reduced or even eliminated, as all data will be readily available.

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] Axboe, Jens. [*PATCH*][*CFT*] *time sliced cfq ver18*. Posted on the linux-kernel mailing list, December 21, 2004. `http://article.gmane.org/gmane.linux.kernel/264676`

[3] Van den Brink, Benno; Almesberger, Werner. *Active Block I/O Scheduling System (ABISS)*. Proceedings of the 11th International Linux System Technology Conference (Linux-Kongress 2004), pp. 193–207, September 2004. `http://abiss.sourceforge.net/doc/LK2004_abiss.pdf`

[4] Cao, Mingming; Ts'o, Theodore Y.; Pulavarty, Badari; Bhattacharya, Suparna; Dilger, Andreas; Tomas, Alex; Tweedie, Stephen C. *State of the Art: Where we are with the Ext3 filesystem*. To appear in the Proceedings of the Linux Symposium, Ottawa, July 2005.

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

[6] Kleen, Andi. [*PATCH*] *x86_64: emulate NUMA on non-NUMA hardware*. Posted on the linux-kernel mailing list, August 31, 2004. `http://article.gmane.org/gmane.linux.kernel.commits.head/38563`

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*