IPSJ Transactions on Advanced Computing Systems Vol. 3 No. 1 61–76 (Mar. 2010)

*Regular Paper*

# A Real-Time File System for Constrained Quality of Service Applications

Damien Le Moal,[†1,†2] Donald Molaro[†3] and Jorge Campello[†3]

The prevalence of multi-tuners, high-definition digital video recorder systems and home networking is increasing the number of simultaneous streams that must be processed by recorder storage devices. Whereas recent hard-disk drives provide enough performance to theoretically handle such workloads, general purpose file systems and I/O schedulers used by operating systems such as Linux do not satisfy the quality of service (QoS) requirements necessary for efficient processing of real-time video streams. In this paper, we introduce the Audio/Video File System (AVFS) composed of a file system and a disk I/O scheduler optimized for handling simultaneous high bit-rate real-time streams. Using a precise QoS measurement method, evaluation results of a Linux implementation of AVFS show that, compared to traditional file systems such as *ext3* and *JFS*, AVFS provides QoS guarantees for real-time streams and more stable performance.

## 1. Introduction

New high-end consumer electronic systems such as high-definition multi-room Digital Video Recorders (DVR) are capable of simultaneously recording digital content from several sources such as cable or over-the-air broadcasting tuners, while at the same time, allowing for local playback of stored content and its streaming to other devices over a home network. Many DVR systems also provide other types of applications such as digital photo storage and viewing, Internet browsing, email, games, etc.

Such entertainment systems are now often implemented using some version of the Linux kernel as an operating system. This approach greatly facilitates the development process and provides a high level of functionalities and performance. In particular, Linux storage subsystems (file system and I/O scheduler) can efficiently harness the high performance of recent hard-disk drives to allow supporting the high load incurred by the simultaneous recording and reading of multiple streams of high-definition video.

However, Linux file systems and I/O schedulers have no inherent ability to distinguish between a real-time application, e.g., playing a movie, and a best-effort task, e.g., viewing a photo. As a consequence, the on-time processing of time critical disk I/O requests cannot be consistently guaranteed, particularly in the presence of best-effort disk accesses, leading to poor Quality of Service (QoS) for real-time video recording and playback applications. In addition to this problem, these traditional storage stacks achieve high throughput at the cost of a high disk utilization rate, leading to higher disk power consumption and operating temperatures.

In this paper, we present the Audio/Video File System (AVFS), a solution to exhaustively address DVR quality and performance problems. AVFS is composed of a file system and a real-time disk scheduler cooperating to provide efficient processing of real-time disk accesses with quality of service guarantees. The proposed file system provides a set of new system calls allowing applications to specify deadlines for real-time requests, thus implementing traffic differentiation. It also implements a block allocation policy resulting in stable performance, independent of the files accessed. The disk scheduler is optimized to deliver high real-time disk throughput with QoS guarantees while minimizing the disk utilization rate through aggressive seek overhead reduction.

This integrated approach is different from traditional Linux implementations where the file system and disk scheduler operate independently from each other. Also, unlike other solutions proposed in the past, AVFS achieves higher performance and QoS using simple solutions that can be implemented in Linux entirely within dynamic loadable kernel modules without requiring any kernel modifications, facilitating its integration in different systems.

The remainder of this paper is organized as follows. Section 2 discusses DVR implementation and problems in more detail. AVFS is described in depth in Section 3, and evaluation results of a Linux implementation are presented in Sec-

†1 Systems Development Laboratory, Hitachi Ltd., Kawasaki, Japan
†2 Graduate School of Informatics, Systems Science, Kyoto University, Kyoto, Japan
†3 Hitachi Global Storage Technologies, San Jose Research Center, San Jose, CA, USA

tion 4. Related work is discussed in Section 5 and Section 6 provides a conclusion to this paper.

## 2. Background

In this section, a typical implementation of a DVR application is presented and used as a basis for discussing DVR performance and QoS requirements. The problems raised by the use of traditional file systems and disk schedulers to achieve these requirements are also discussed.

### 2.1 Typical DVR Implementation

A typical implementation of a multi-stream DVR system can be roughly split into several software layers as shown in **Fig. 1**.

At the user level, stream applications manage input streams (recording of video data obtained from a tuner or a network feed) and output streams (video local playback using a hardware decoder or transmission over a network) by issuing read and write I/O requests to the file system. In this model, all I/O requests issued by stream applications have completion time constraints to ensure smooth recordings, playbacks or network streaming of video content. Applications without real-time requirements may also be executed simultaneously. These applications are classified as best-effort and may also execute file I/O operations.

To accommodate the different access models (block based or character based) and response times of the devices being used, stream I/O operations are processed using buffers to temporarily store video data. If one were to examine the amount of data in the buffers of real-time streams (i.e. buffer level) over time, a pattern as shown in **Fig. 2** would be observed. In read streams (such as video playback) the buffer level depletes as the video data in the buffer is used. At some point in time (submission time), the application has the opportunity to issue an I/O request to partially refill the buffer. Some time later (service time), the request completes and the buffer level increases. The reverse applies for write streams (e.g., a video recording).

At the time a request is submitted, a stream application can calculate a deadline for the submitted request. In the case of a constant bit rate video, dividing the buffer size by the rate at which it is being depleted (or filled) gives the deadline. In variable bit rate streams, the deadline can be calculated by finding the number of video frames that the buffer contains and the frame display rate. Real-time request deadlines may also be calculated using video frame information often provided by digital tuners during video recording. The time interval between a request completion time and its deadline is the I/O completion margin.

### 2.2 Measuring DVR Quality of Service

A well behaved DVR system, or in other words a DVR system providing good quality of service, is one that processes all stream I/O requests within their deadlines to avoid buffer under-run in output streams and input stream buffer overflows, even in the presence of competing best-effort disk accesses.

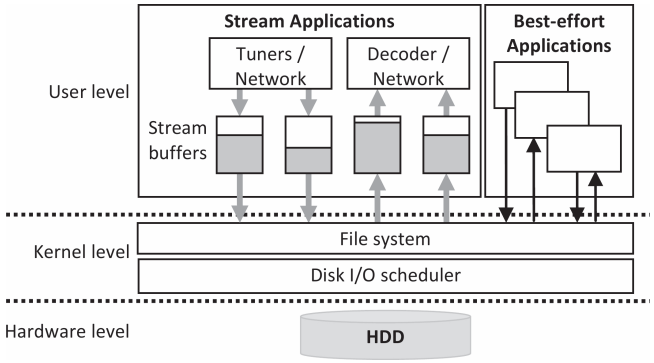


**Fig. 1** Typical DVR software architecture.

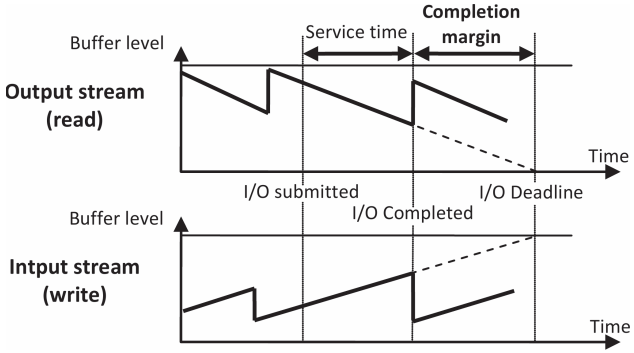


**Fig. 2** I/O request deadline relation to a stream buffer state.

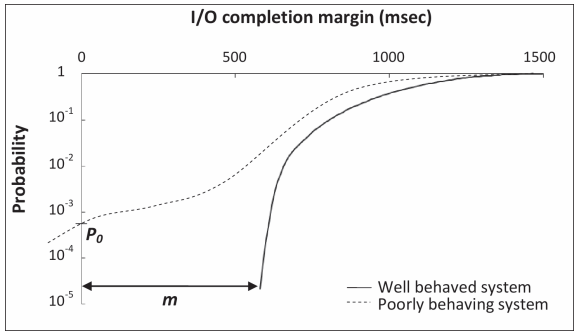


**Fig. 3** Cumulative distribution function (CDF) for real-time I/O request completion margin. The y-axis shows the probability that a real-time request will be completed within a margin lower or equal to a particular value (x-axis).

The QoS level of a DVR can be precisely defined by utilizing the rate of deadline misses when handling real-time requests. This rate can be calculated statistically by keeping track of the submission time, completion time and deadline of real-time requests. For a sufficiently large set of data, one can obtain an estimate of the probability distribution function (PDF) of the completion margins of real-time requests. Integrating this PDF leads to the cumulative function (CDF) for the completion margins (**Fig. 3**).

The CDF indicates the probability (y-axis in Fig. 3) that a real-time request will be completed within a margin lower or equal to a particular value (x-axis in Fig. 3). As the completion margin is negative only for requests with missed deadlines, the probability given for a 0 margin (point $P_0$ in Fig. 3) is equal to the probability of any one request missing its deadline. By knowing the average amount of data processed in each request and the bit rate of the observed streams, it is possible to show that the system meets a specific QoS level, or in other words meets requirements such as "less than one glitch per hour".

The positioning of the tail of the CDF curve along the time axis also reveals the overall system performance. If the tail of the CDF curve is well away from the Y-axis, e.g., the distance $m$ in Fig. 3 is large, the system is over-buffering and the system designer can safely reduce the size of stream buffers without any impact on the observed quality of the streams. If on the other hand the curve shows margins close to 0 but still positive, the system is meeting all real-time request deadlines without much room for the system to support unexpected events such as a lengthy disk error recovery procedure when a defective disk sector is accessed.

### 2.3 DVR Requirements and Implementation Problems

The QoS metric introduced in the previous section shows that a DVR only needs to ensure that real-time I/O requests complete within at least the desired margin. An implementation of a DVR storage subsystem should thus first provide a performance level reasonably high enough to allow the on-time completion of real-time I/O requests for several streams of high-definition video. Such performance level must also be achieved even in environments where there is a mixture of real-time and best-effort disk accesses. That is, the implementation must be able to differentiate real-time and best-effort requests to apply an adapted scheduling policy and achieve QoS guarantees for real-time accesses.

Another quality aspect that must be provided is stability: the performance level should be stable enough over time so that the system can consistently provide the same level of functionality to its user.

Finally, the implementation should also fit reasonably well into the typical application model presented previously so that it can be easily adopted by the relevant development community.

Modern file systems and I/O schedulers in Linux, combined with recent hard-disk drives, generally provide a throughput sufficiently high to process several streams of high-definition video simultaneously. However, I/O system calls in Linux are based on the POSIX specification which lacks a means to specify the nature of an I/O operation (real-time or not) as well as a method to attach a time constraint (deadline) to I/O requests. As a result, I/O operation classes cannot be differentiated in order to implement QoS guarantees for real-time applications. It has also be shown that Linux file systems, while not very sensitive to file fragmentation due to the repeated deletion and creation of files, exhibit variations of up to 30% in file access performance [1]. For a DVR, this can lead to a different maximum possible number of streams that can be processed over time, or in other words to unstable and unpredictable performance.

## 3. The Audio/Video File System

The Audio/Video File System (AVFS) was developed to exhaustively address the DVR storage stack requirements presented in the previous section. It uses simple methods integrated to achieve efficient processing of multiple high-definition video streams.

### 3.1 Overview

AVFS simultaneously addresses performance as well as QoS issues using two main components, as shown in **Fig. 4**. First the A/V file system provides support for new I/O system calls (A/V I/O library) allowing assigning deadlines to real-time requests, while preserving the standard POSIX set of system calls for best-effort I/O operations. The A/V file system also improves file access performance stability by distributing file data blocks across the entire disk partition. The second main component of AVFS is a disk scheduler called the traffic mixer. The traffic mixer uses real-time request deadlines to implement QoS guarantees while increasing performance of disk accesses through disk seek overhead reduction using batch processing of real-time requests.

AVFS can coexist with a standard file system on a different partition of a single disk drive. In such cases, the traffic mixer treats all disk I/O requests issued by the standard file system as best-effort traffic.

The Linux implementation of AVFS uses dynamic loadable modules. In addition to avoiding the need for any kernel modification, this implementation approach also simplifies the integration of AVFS into the various Linux distributions provided by DVR hardware vendors.

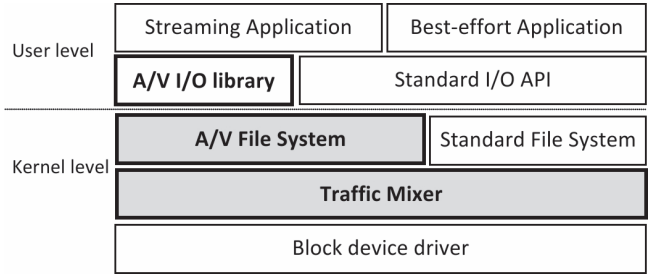


**Fig. 4** Organization of AVFS components.

### 3.2 A/V File-System

The A/V file system is a journaling file system which shares many features with the traditional UNIX file system such as its inode metadata structure. The differences mainly reside in its partition format which separates metadata and data blocks into distinct regions, its data block allocation policy using very large blocks distributed over the entire disk partition, and its support for the real-time I/O system calls implemented by the user level A/V I/O library.

#### 3.2.1 Partition Format

As shown in **Fig. 5**, an AVFS disk partition is separated into three regions: the primary metadata region, the data region, and the backup metadata region.

The primary metadata region is organized as a set of 4 KB blocks used to store metadata. It consists of the super block (SB) for describing the file system format, a set of contiguous blocks for the journal, blocks used as bitmaps for controlling the allocation state of metadata blocks, data blocks and inodes, and blocks containing inode data structures. The remaining of the metadata blocks are dynamically used for storing the data block mapping of files and directory entries.

The backup metadata region is a copy of the primary metadata region, with the exception of the journal blocks. This copy is maintained through AVFS journaling and is used to improve the robustness of the file system against sector corruption in the primary metadata region.

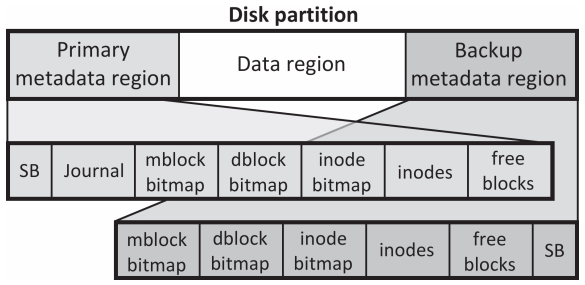


**Fig. 5** Disk partition format with the A/V file system.

### 3.2.2 Data Block Management

Since modern disk drives are organized into zones whose sector density decreases toward the disk center, the disk drive data transfer rate is not uniform but instead decreases towards the center of the disk. This leads to a large deviation in I/O processing time depending on the sectors accessed and thus to non-predictable maximum performance.

To solve this problem, the A/V file system distributes data blocks of a file across the entire data region to implement a uniform and thus predictable performance for sequential accesses to files. This block allocation policy is called Zone-Round-Robin (ZRR). The ZRR policy allocates data blocks to files using groups of contiguous data blocks (zones). Data block allocation for a file is done by first choosing a zone, then a data block within the zone. Zones are used in a round-robin manner, that is, consecutive data blocks of a file are chosen from consecutive zones. The first free data block of a zone is always chosen as the allocation candidate. The first data block of a file is always allocated from the zone following the one last used for an allocation. The zone used for the first data block allocation after a file system formatting is chosen randomly.

To simplify implementation, the data region is divided up into zones without any consideration of the physical layout of the disk sectors, tracks and cylinders. That is, data block zones are generally not aligned on disk tracks or cylinders. The size and the number of zones are computed using the size of the data region so that at most 32 zones are created with each zone size no smaller than 16 GB. These values were arbitrarily chosen. The study of the impact on performance of different zone division methods is left as a subject for future study and not discussed in this paper.

An example of this allocation scheme is shown in **Fig. 6**: the data region is divided into $n$ zones of $m$ blocks. The file system is initially empty and data blocks for file A are allocated starting from the zone $i$. Since the file system is empty, the first block of each zone is allocated to file A resulting in the sequence shown. Block allocation for the files created following A are similarly allocated so that, roughly speaking, all files are uniformly spread out across the whole data region. Sequential accesses to files thus result in all zones being traversed sequentially and cyclically, accessing one or more blocks in each zone visited. As

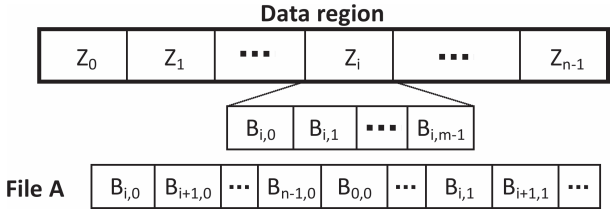


**Fig. 6** A/V file system ZRR block allocation policy.

a consequence, the total access time for a file is almost totally independent of its block locations but is simply proportional to its size with an almost constant seek time between blocks and almost constant average transfer time of blocks. In other words, the ZRR policy achieves a stable average I/O processing time, leading to more stable and predictable overall disk performance.

To mitigate the performance penalty due to the introduction of disk head seeks between logically sequential requests in different blocks in consecutive zones, that is, to avoid excessive file fragmentation, the A/V file system uses large data blocks. The default size for data blocks is 4 MB, which corresponds to about 1.7 seconds of ATSC MPEG2 high-definition video with a bit rate of 19.39 Mbps. this default value can be changed during disk formatting to any value that is a power of 2 higher than 4 KB and up to 4 GB.

Beyond stabilizing access performance, this large data block management strategy has also the advantage of reducing the amount of metadata required to store a file data block mapping on disk. As was shown with the Conquest file system [2], this can further improve disk performance.

### 3.2.3 Real-time I/O System Calls

AVFS real-time I/O system call functions are implemented as an application level library (A/V I/O library in Fig. 4) using the standard `ioctl` system call. The functions provided, shown in **Table 1**, are similar to the POSIX `aio` interface.

As shown in **Fig. 7**, AVFS real-time I/O system call functions are used in the same way as their POSIX equivalent (**Fig. 8**). To use these functions, the first step is to set up an asynchronous I/O descriptor (`aio` variable). The main difference between the two sets of functions is the `aio_deadline` field in the

**Table 1** AVFS real-time I/O system call functions.

| Function | Description | POSIX equivalent |
|---|---|---|
| avfs_asyncio_start | Start an asynchronous I/O request | aio_read<br>aio_write |
| avfs_asyncio_check | Wait or check the completion of an asynchronous I/O request | aio_suspend<br>aio_error |
| avfs_asyncio_cancel | Cancel the processing of a started asynchronous I/O request | aio_cancel |
| avfs_asyncio_poll | Poll a set of files for the completion of asynchronous I/O requests | poll |

```
/* AVFS aio example */

avfs_asyncio_t aio;
...
/* Setup aio */
aio.aio_fildes = fd;
aio.aio_offset = ofst;
aio.aio_buf = buf;
aio.aio_nbytes = size;
aio.aio_deadline = dl;

/* Start aio */
avfs_asyncio_start(&aio, AVFS_READ);

/* Do some other processing */
...

/* Wait for aio completion */
avfs_asyncio_check(&aio, AVFS_WAIT);
if ( aio.aio_ret < 0 ) {
    /* I/O error */
}

/* Use data */
...
```

**Fig. 7** Simple example of a typical use of AVFS real-time I/O system calls.

```
/* POSIX aio example */

struct aiocb aio;
...
/* Setup aio */
aio.aio_fildes = fd;
aio.aio_offset = ofst;
aio.aio_buf = buf;
aio.aio_nbytes = size;


/* Start aio */
aio_read(&aio);

/* Do some other processing */
...

/* Wait for aio completion */
aio_suspend(&aio, 1, 0);
if ( aio_return(&aio) < 0 ) {
    /* I/O error */
}

/* Use data */
...
```

**Fig. 8** Simple example of a typical use of the POSIX asynchronous I/O system calls.

`avfs_asyncio_t` data structure. This field allows an application to specify a deadline in absolute system time, indicating that the request is a real-time one. This differentiation is not possible with the POSIX interface as such a field is not available in the `aiocb` data structure. The next steps of the processing are starting the I/O request and checking its completion. These steps are done in a very similar manner with the two interfaces.

For AVFS asynchronous I/O requests, the deadline indicated by the `aio_deadline` field is passed down to the traffic mixer by the A/V file system using the `bi_private` field of Linux block I/O data structures (`struct bio`) generated to process the application I/O request. This information is used by the traffic mixer to differentiate real-time requests from best-effort ones. Block I/O requests with no deadline specified (i.e. the `aio_deadline` field is set to 0) are assumed by the traffic mixer to be best-effort. The same also applies to any block I/O request generated by the execution of a standard POSIX I/O system call.

### 3.3 Traffic Mixer

The traffic mixer is a real-time disk I/O scheduler. Its goal is to mitigate the adverse effects of the periodic nature of real-time requests on the spatial locality of disk accesses, that is, on the degradation of disk seek overhead due to the processing of non-sequential requests directed to different files. This is achieved by delaying the processing of real-time requests according to their deadlines and to their estimated processing time so that several sequential I/O operations to a single file can be accumulated in the scheduler queue. This approach virtually creates a request pattern similar to a fast file access where sequential I/O are issued quickly one after another. Such a pattern can be processed more efficiently using a typical elevator type policy (e.g., C-SCAN [3]), which drastically reduces seek overhead and improves performance.

#### 3.3.1 Overview

The traffic mixer is organized as shown in **Fig. 9**. Received block I/O requests first go through an admission step to separate real-time and best-effort requests into different queues. A request with a specified deadline is treated as a real-time request. All other requests are treated as best-effort.

Both real-time and best-effort request queues are maintained in their execution order that is, the order in which requests can be executed as efficiently as possible, i.e., with the least amount of seek between requests. The optimal execution order in general varies according to the disk drive model, its physical block layout,

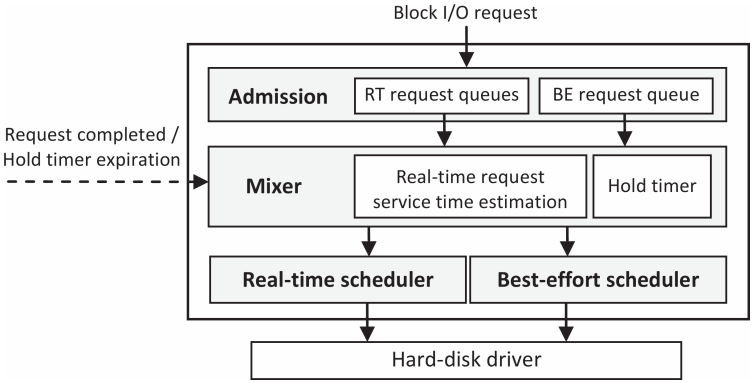


**Fig. 9** A/V traffic mixer organization.

etc. The traffic mixer considers only a simplistic disk model and uses increasing logical block address (LBA) ordering as the execution order (equivalent to a C-SCAN policy). Real-time requests are also queued into another list maintained in deadline order.

Once requests are differentiated and queued, the traffic mixer estimates the service time of each real-time request assuming the requests are executed in LBA order. Depending on the result of this estimation, the mixer either sets the hold timer to delay the processing of pending real-time requests, activates the real-time scheduler to start processing all pending real-time requests, or activates the best-effort scheduler to execute a single best-effort request. The traffic mixer always dispatches requests to the disk driver one at a time.

In its current implementation, the traffic mixer does not provide a bandwidth reservation mechanism. Scheduling decisions are made dynamically upon request arrival and completion. A possible consequence is that the system might become overloaded with real-time traffic, causing the real-time scheduling to fail. To avoid this problem, a DVR system designer can rely on the performance stability provided by the A/V file system and use the QoS measurement method presented in Section 2 to verify during a DVR development phase if a target workload is within the maximum performance achievable by the system.

#### 3.3.2 Real-time Request Service Time Estimation

The service time of real-time requests is estimated using a simple disk model. This model is based on the disk rotational period $T_r$, its average seek time $T_k$, its maximum and minimum data transfer speed $D_{tmax}$ and $D_{tmin}$ and the largest LBA of the disk $L_{max}$. The model assumes a linear variation of the disk transfer speed between the minimum and maximum LBA. The service time $T_S(R)$ of a real-time request $R$ of size $S(R)$ and of starting LBA $L(R)$ is estimated as follows.

$$T_S(R) = \alpha \left( T_k + \frac{T_r}{2} \right) + \frac{S(R)}{D_{tmax} - \frac{(D_{tmax} - D_{tmin}) L(R)}{L_{max}}} \quad (1)$$

The request service time estimate is in other words just the average seek time plus the average rotational latency plus the data transfer time. The seek time factor $\alpha$ is equal to 1 if the estimated request is not sequential with respect to the previous one estimated in execution order. Otherwise, $\alpha$ is set to 0 to reflect the absence of disk head seek.

A more precise estimator could be designed using more detailed disk models [4]. However, the estimator used here, while crude, has the advantage of using only a small set of parameters that can be easily obtained from the disk specifications or directly measured. Despite its simplicity, this model also has the advantage of lowering the probability of underestimation to reasonably small level.

#### 3.3.3 Mixer Algorithm

The role of the mixer is to control execution of the real-time and best-effort schedulers so that real-time requests are processed before the expiration of their deadlines and the average seek distance between consecutive real-time requests in execution order is minimized. The mixer does this by delaying the processing of real-time requests until the time $H$ at which it must start executing all pending real-time requests to meet their deadlines. The time $H$ is calculated as follows using the index set $e_0, e_1, \cdots, e_{N_{R-1}}$ to represent the sequence of $N_R$ real-time requests in execution (LBA) order, with $T_D(R)$ being the deadline of request $R$ and $m$ the desired completion margin (see Fig. 3).

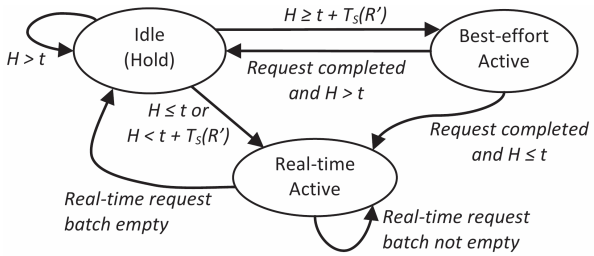


**Fig. 10** Traffic mixer scheduling states.

$$H_j = T_D(R_{e_j}) - \left( m + \sum_{i=0}^{j} T_S(R_{e_i}) \right) \tag{2}$$

$$H = \min_{j=0 \ldots N_R - 1} H_j \tag{3}$$

Here $H_j$ represents the latest time at which LBA order execution of the request $R_{e_j}$ and of its predecessors can start without missing its deadline within the margin $m$. The minimum $H_j$ for the entire set of $N_R$ real-time requests, i.e. $H$, thus gives the latest possible time to activate the real-time scheduler to execute all requests in LBA order so that a completion margin of at least $m$ is achieved for all requests.

Using $H$, the mixer controls activation of the real-time and best-effort schedulers as shown in **Fig. 10**. At the current time $t$, if $H > t$ then the mixer delays execution of pending real-time requests (hold state). If this condition becomes false, the mixer activates the real-time scheduler to process all pending real-time requests [⋆1]. When both best-effort and real-time requests are present and the real-time scheduler is not activated, if $H \geq t + T_S(R')$, where $R'$ is the next best-effort request in the queue, then $R'$ is executed and the set of real-time requests remains schedulable. If the test fails the mixer activates the real-time scheduler. In the absence of real-time requests, the best-effort scheduler is always activated immediately upon arrival of a new best-effort request.

The delayed processing of real-time requests introduced by the mixer implies that efficient processing of real-time I/O operations can be achieved only using the asynchronous I/O model (as provided by the A/V I/O library). Using traditional blocking I/O functions for real-time requests would result in the introduction of a large apparent I/O processing time, and result in potential stream buffer underrun or overflow (i.e. video quality problems).

### 3.3.4 Real-time Scheduler Policy

As discussed in the previous section, the mixer activates the real-time scheduler to process all $N_R$ real-time requests in the LBA ordered queue. Therefore, when activated by the mixer, the real-time scheduler removes all real-time requests in the execution ordered queue to form a single batch of requests. Since $H$ is calculated using the execution order of requests with deadlines reduced by the margin $m$, the real-time scheduler assumes that all requests in the batch can be executed in LBA order to improve disk throughput. However, a request may still miss its deadline if an unexpected event such as a disk error recovery results in a large difference between the estimated service time and actual execution time of previous requests.

To address this problem, the real-time scheduler operates depending on the current time $t$ and on the earliest deadline $t_d = T_D(R_d)$ of the requests in the batch. First, if $t_d \leq t$, i.e., $R_d$ missed its deadline, the request is not processed and immediately terminated as failed. If $t_d > t$ but $t_d \leq t + e$, where $e < m$ is a parameter to identify requests to be urgently scheduled, then $R_d$ and all succeeding requests having urgent deadlines are scheduled in an earliest-deadline-first (EDF) manner until $T_d(R) > t + e$ is satisfied for all remaining requests $R$.

If $t_d > t + e$ but $t_d \leq t + m$, meaning that the request with the earliest deadline cannot be executed without missing the desired completion margin $m$, then LBA order processing is only applied to the subset of requests $\{R_j | T_D(R_j) < t_d + m\}$. Otherwise, that is if $t_d > t + m$ as expected and obtained in most cases, the real-time scheduler starts executing the request $R_{e_0}$ and its successors in LBA order.

As shown in **Fig. 11**, this policy implicitly defines three different scheduling windows defining the sets of requests considered for execution by the real-time

⋆1 Strictly speaking, the processing of real-time requests could be interleaved with best-effort requests when better candidates in terms of seek can be found, and all remaining real-time requests have a large enough $H_j$ to tolerate additional delays. However, the existence of such best-effort requests is unlikely, leading to only a negligible improvement in the response time of best-effort requests. The entire set of real-time requests is thus simply passed as a whole to the real-time scheduler.

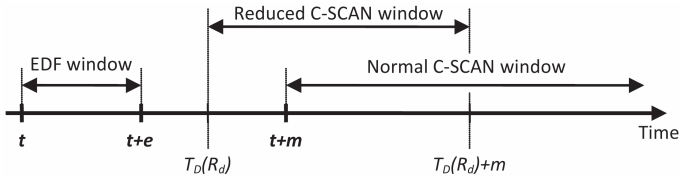


**Fig. 11** Scheduling windows of the traffic mixer real-time scheduler.

scheduler. The EDF window is used only if some real-time requests have a deadline below $e$. The reduced C-SCAN window is used for requests with a deadline within the earliest deadline plus the desired completion margin $m$. The normal case C-SCAN window is used for all other cases. If the real-time workload applied to the system is within the system maximum, the EDF and reduced C-SCAN windows will be used only if the processing time for some requests exceeded the estimate or if an application issued some urgent requests.

Any real-time request received while the real-time scheduler is active is only added to the current batch if its deadline is earlier than the latest deadline of the requests in the current batch. Otherwise, this new real-time request is left pending until a new batch processing is started.

#### 3.3.5 Best-effort Scheduler Policy

Best-effort requests are always executed in increasing LBA order, starting from the end LBA of the last request executed (either real-time or best-effort). Since the processing of best-effort requests may be interrupted to execute real-time requests, the LBA variation may not be monotonic, potentially resulting in requests starvation. The best-effort scheduler takes care of such cases by managing the age of its pending requests, ensuring that older requests are processed with a higher precedence over new ones.

## 4. Experimental Results

This section presents evaluation results from implementing AVFS on Linux and using a DVR workload generator application which mimics the simultaneous recording and playback of multiple streams of high-definition video. In order to precisely determine the relative contribution to performance and QoS of the A/V file system and of the traffic mixer, the A/V file system was used alternatively in combination with Linux *cfq* scheduler [5] and with the traffic mixer. Results obtained with these two configurations were compared with those obtained when using Linux *ext3* [6] and *JFS* [7] file systems in combination with the *cfq* scheduler.

### 4.1 Evaluation Environment

All experiments were performed using a 250 GB SATA hard-disk drive (3.5 inches form factor, 7200 rpm, 8 MB buffer, 2 platters and 4 heads). The disk write cache was disabled for all experiments and the read-ahead functionality was kept enabled. The disk maximum read throughput was measured at 560 Mbps. This disk was attached to a mini ITX PC motherboard equipped with a VIA C3 (1 GHz) CPU and 128 MB of RAM. A standard (unmodified from official release) Linux kernel version 2.6.24 was used for all experiments.

The DVR workload generator application used was a multi-threaded application using one thread per stream. The workload generated for each stream corresponds to either the recording or playback of ATSC MPEG2 video files with a bit-rate of 19.39 Mbps. The I/O operations for streams were executed using POSIX `aio` functions for *ext3* and *JFS* evaluation and AVFS asynchronous I/O operations for AVFS measurements. The traffic mixer scheduler was set up with a desired completion margin (parameter $m$) for real-time requests set to 500 milliseconds. The urgent request threshold (parameter $e$) was set to 100 milliseconds.

In all experiments, stream I/O operations were 512 KB direct I/O, representing 217 milliseconds of high-definition video data. This led to applications issuing I/O requests with a 217 milliseconds cycle, thus making the overhead due to execution of system calls negligible. An application was allowed to issue simultaneously at most 8 I/O requests per stream, representing a per stream buffer of 4 MB (i.e., 1.73 seconds of video data). Each experiment was executed for one hour.

The workloads tested are identified using the notation $NrMw$ where $N$ is the number of read streams (playbacks) and $M$ is the number of write streams (recordings).

### 4.2 Performance Measurements

This section presents evaluation results of the streaming performance achieved by different configurations. Performance was measured by observing the number of streams that can be simultaneously processed in real-time without deadline

miss. The stability and efficiency of the tested configurations were also quantified using the disk utilization rate (i.e. the percentage of time the disk is busy processing I/O requests).

### 4.2.1 Real-time Streaming Performance

In this experiment, using a randomly chosen set of files from a file system filled up to 80% of capacity, the I/O completion margin CDF for workloads of 2, 4, 6 and 8 read streams and 0, 2, 4 and 6 read streams combined with 2 write streams was measured. Results for *ext3* and *JFS* used in combination with the *cfq* scheduler are shown in **Fig. 12**. Results for the A/V file system used in combination with the *cfq* scheduler (AVFS-cfq case) and the traffic mixer scheduler (AVFS-tm case) are shown in **Fig. 13**.

Referring to Fig. 12, *JFS* clearly processed all I/O requests quickly, completing the processing of all requests well within the desired margin. It can be observed however, that as write streams are introduced, the tail of the CDF curves shift left, with the heaviest workload ($6r2w$) clearly showing the lowest completion margins. On the other hand, *ext3* failed to maintain QoS for the two workloads of 8 streams ($8r0w$ and $6r2w$). These two cases showed a deadline miss probability $P_0$ of 0.001 and 0.003 respectively. This corresponds to a video glitch every 28.9 min and 9.6 min in average per stream.

As shown in Fig. 13, combining the A/V file system with the *cfq* scheduler (AVFS-cfq case) resulted in completion margins very similar to the margin obtained with *JFS*. The desired QoS constraint of 500 milliseconds was maintained for all test cases. The use of the traffic mixer (AVFS-tm case) also led to the same results with the tail of all CDF curves converging toward the desired completion margin.

These results show that the performance level that can be obtained with the A/V file system, regardless of the scheduler being used, is similar to that of *JFS*, or in other words has the ability to simultaneously support at least 8 streams of high-definition video. Here, *ext3* provided lower performance with a maximum of only 6 streams.

### 4.2.2 Performance Stability

In order to observe the performance stability of the different configurations, this experiment repeatedly measured the disk utilization rate for the $4r2w$ workload.

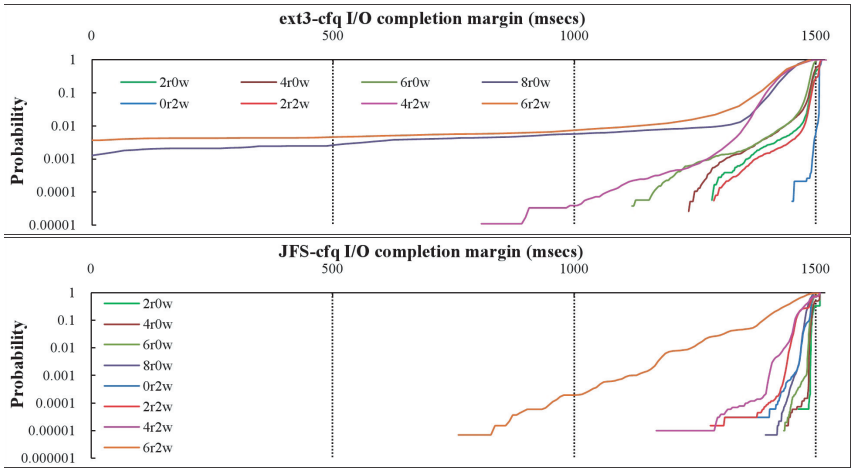


**Fig. 12** Real-time request completion margin CDF for various workloads using *ext3* and *JFS* with the *cfq* scheduler.

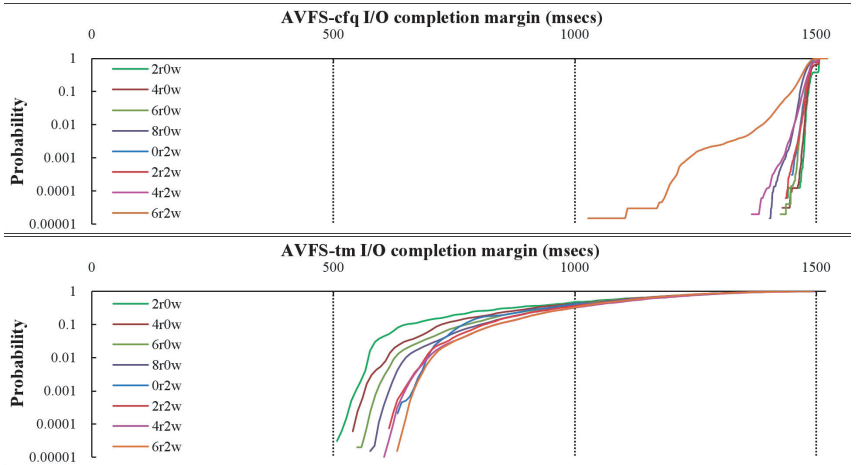


**Fig. 13** Real-time request completion margin CDF for various workloads using AVFS in combination with the *cfq* scheduler (AVFS-cfq case) and the traffic mixer scheduler (AVFS-tm case). Completion margins of the AVFS-tm case are lower due to the delayed processing of real-time requests, but not lower than the desired completion margin (parameter $m$) set to 500 msecs in this test.

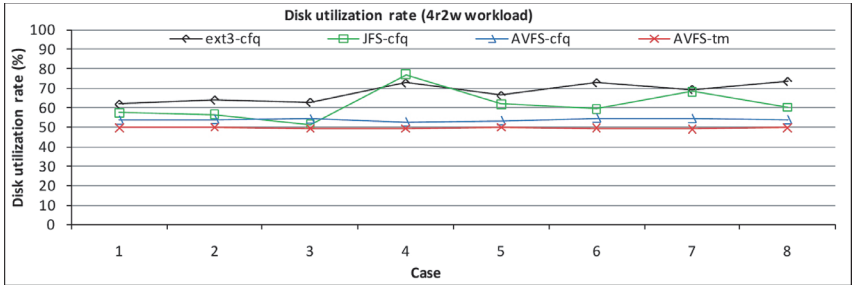


**Fig. 14** Disk utilization rate stability for different files ($4r2w$ workload).

For each measurement, a different set of files was utilized for the read streams. The files created by the two write streams of this workload were deleted at the end of each test and another test started without reformatting the file systems.

As shown in **Fig. 14**, both *ext3* and *JFS* exhibited a large variation in the disk utilization rate depending on the set of files being used. This instability shows that the $4r2w$ workload may potentially result in deadline misses for some file combinations. In other word, the performance is not stable and thus the ability to process this workload without deadline miss not predictable.

On the other hand, the A/V file system combined with the *cfq* scheduler showed a more stable disk utilization rate. This rate was also lower than that obtained with *ext3* and *JFS*. The AVFS-tm configuration also led to the same result with a slightly lower (improved) disk utilization rate.

This higher stability of the disk utilization rate obtained with AVFS, regardless of the scheduler being used, can be explained by the data block management policy of the file systems tested. As *ext3* and *JFS* allocate data blocks to files sequentially, simultaneous access to different files result in a seek distance between I/O requests that is dependent on the relative position of the files on disk. As the time necessary to process a request increases significantly with the seek distance [8),9)], lower seek distances on average lead to lower disk utilization. In other words, if the accessed files are close together, seek can be reduced, resulting in a lower overall disk utilization rate. Simultaneous accesses to files far apart on disk render an opposite result.

The distribution of a file data blocks across the disk done by AVFS solves this

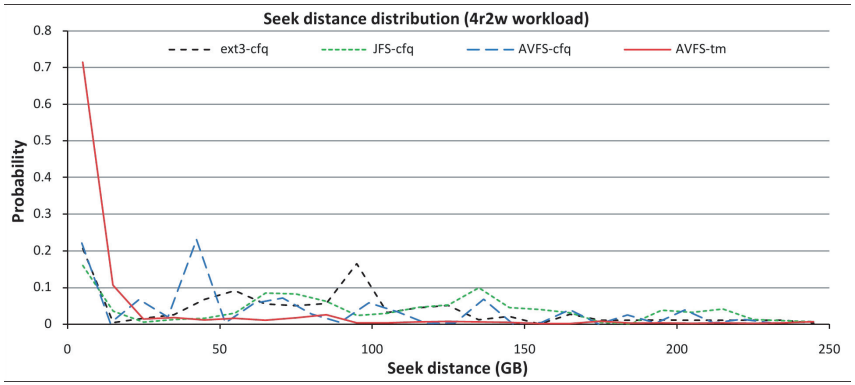


**Fig. 15** Seek distance distribution of consecutive requests for the $4r2w$ workload.

problem, leading to a disk utilization rate that is only dependent on the number of streams and not on the files being used. AVFS data block management therefore provides more stable and predictable performance.

#### 4.2.3 Efficiency

The results presented in Fig. 14 also clearly showed that using the traffic mixer scheduler yielded the lowest disk utilization rate. As for the stability results, this improvement can also be explained by looking at the distance on disk (i.e. amount of seek) between consecutively processed requests (**Fig. 15**).

The use of *ext3* and *JFS* led to a majority of requests separated by seek distances in excess of 50 GB. Since the *cfq* scheduler quickly processes requests in order to minimize response times, the cyclic nature of stream I/O operations results in a scheduling queue generally containing requests directed to different files, and rarely to a sequence of contiguous requests within a file. This characteristic of *cfq* exerts less of an effect on AVFS because of the distribution of data blocks across the disk, leading to a majority of consecutively executed requests separated by a seek distance below 50 GB.

On the other hand, the use of the traffic mixer with AVFS resulted in an I/O request processing sequence with an average lower seek distance or namely to more efficient processing. As the delayed processing of real-time requests by the traffic mixer allows accumulation of sequential requests within a file, more than

70% of the processed requests were separated by a seek distance lower than 5 GB.

#### 4.2.4 Stability Over Time

Previous results not only showed that AVFS performance is stable regardless of the set of files being used, but also that AVFS processes workloads more efficiently with a lower disk utilization rate. To verify that these results are also achieved independently of the age of the file system, that is, on the repeated deletion and creation of files, the workloads applied in Section 4.2.1 are repeatedly executed at increasing file system ages.

The age of a file system is defined here as the total amount of video time recorded without reformatting the file system. The aging process repeatedly deletes files and creates new ones by executing write streams, emulating the repeated recording of video files. The 0 hour age case corresponds to a file system filled up to 80% of its capacity after a fresh formatting. At each measurement point, the set of files used for real-time streams is chosen randomly. Results are shown in **Fig. 16**.

Compared to *ext3* and *JFS*, the disk utilization rate with AVFS did not vary significantly with the file system age. The observed variation over time was smaller than 1% for all workloads whereas the maximum difference reached 36% for *JFS* and 15% for *ext3*. Disk utilization rates with AVFS were also consistently lower for almost all workloads. Only *JFS* performed better for the $2r0w$ case with an average disk utilization rate over time of 9.5% against 14% for AVFS. For higher workloads such as $8r0w$, AVFS resulted in a disk utilization rate lowered to 57% from an average of 88% for *ext3* and 82% for *JFS*.

### 4.3 QoS Measurements

This section presents evaluation results showing the quality of service achieved by all file system configurations.

#### 4.3.1 Effect of Metadata Traffic

This experiment measured the effect of metadata accesses by the file system on real-time streams by deleting a randomly picked video file while the $2r0w$ workload was executed. To characterize the effect of the file deletion, the amount of data in the buffer of one of the read streams was observed. Measurement results are shown in **Fig. 17**. The file deletion processing interval is shown in gray.

Neither *JFS* nor AVFS (regardless of the scheduler being used) were affected by the large file deletion because the amount of metadata requiring processing is small. The file deletion took less than one second and did not cause any noticeable change in the buffer level of the observed read stream. In contrast, the data block management scheme used by *ext3* resulted in a large amount of metadata block accesses to process the file deletion, causing several buffer underruns. The overall deletion operation was also slower and took 29 seconds to complete.

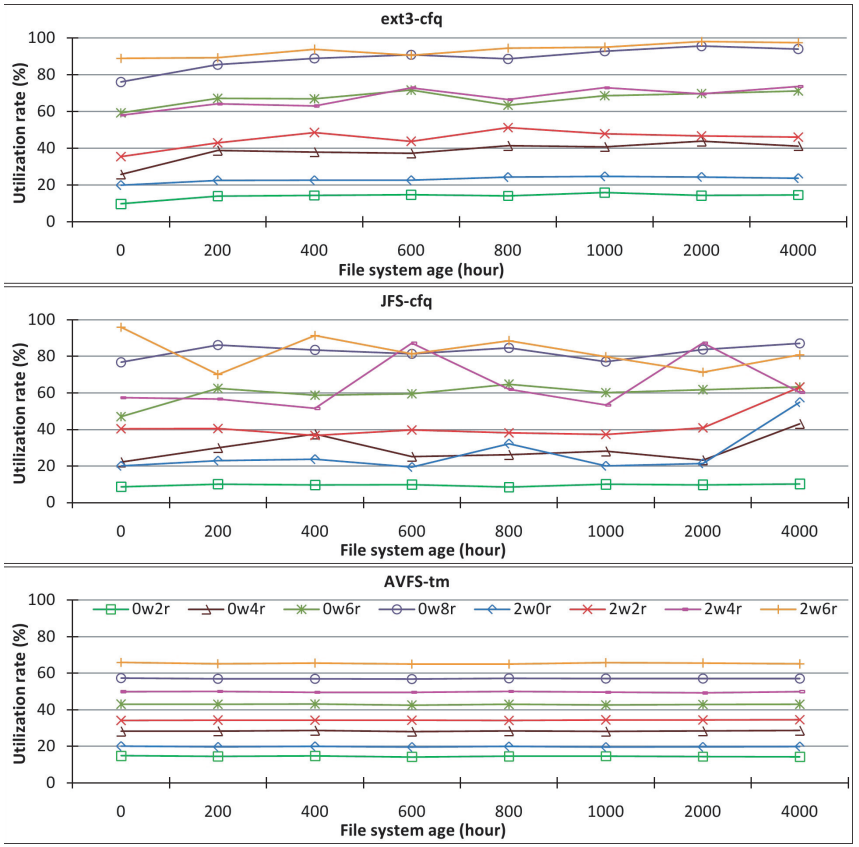


**Fig. 16** Disk utilization rate stability over time (same workloads as for Fig. 12 and Fig. 13).

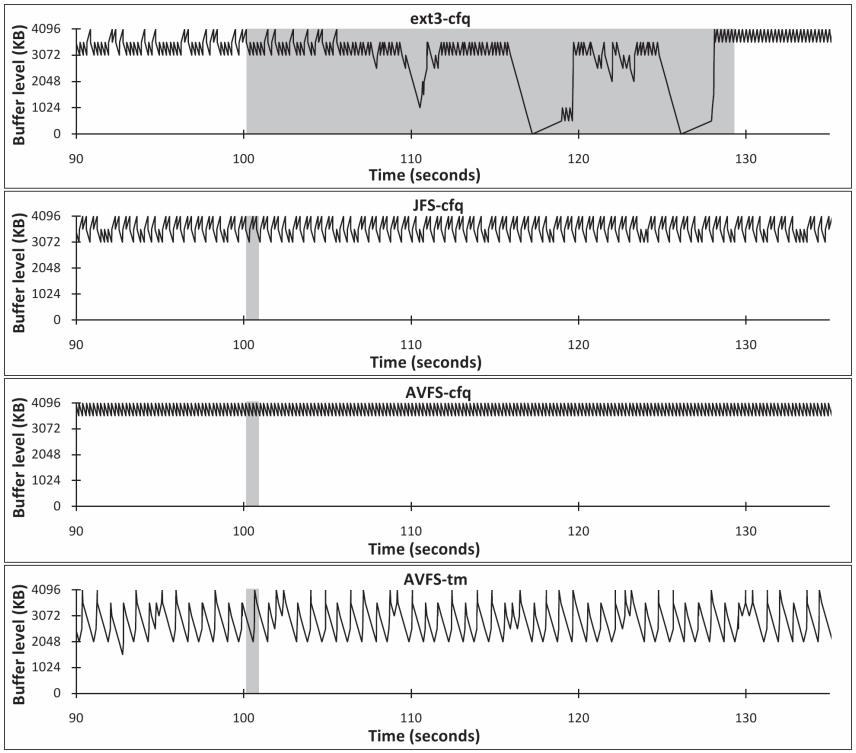


**Fig. 17** Effect of a file deletion on the buffer level of a read stream of the $2r0w$ workload. Buffer level fluctuation in the AVFS-tm case is more important due to the traffic mixer delayed processing of real-time requests.

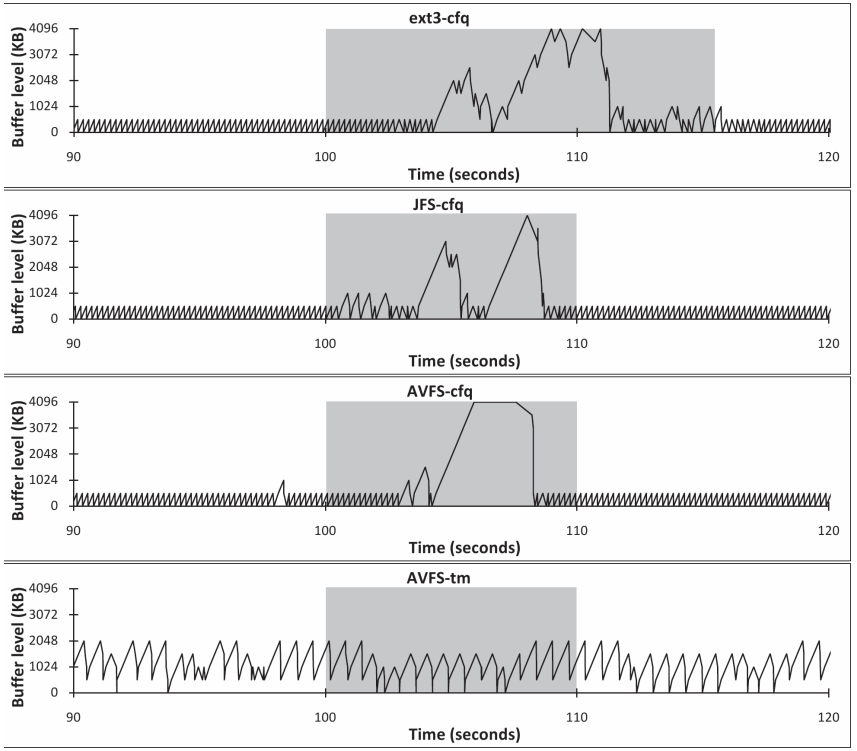


**Fig. 18** Effect of best-effort traffic on the buffer level of a write stream of the $2r2w$ workload. Buffer level fluctuation in the AVFS-tm case is more important due to the traffic mixer delayed processing of real-time requests.

### 4.3.2 Effect of Simultaneous Best-effort Traffic

Under the $2r2w$ workload, this experiment incorporated best-effort accesses by writing a set of 100 pictures to the disk. Picture files averaged 1.3 MB in size, with the entire set of files representing 135 MB of data. The buffer level of one of the write stream (recording) was observed to characterize the effect of the best-effort traffic. Results are shown in **Fig. 18**.

The use of the *cfq* scheduler, independently of the file system being used, resulted in buffer overflows for the observed write stream during execution of the best-effort write operations. As *cfq* does not differentiate processing of real-time and best-effort requests, unacceptable delays in the processing of real-time I/O requests are introduced. The effective bandwidth for writing the entire set of files was comparable for AVFS and *JFS* at 108 Mbps. Here, *ext3* did not perform as well and achieved a rate of only 72 Mbps.

The traffic mixer with AVFS on the other hand processed real-time requests without causing any buffer overflow, which demonstrates the effectiveness of the scheduling policy implemented. The traffic mixer use also caused no significant degradation in the processing speed of the best-effort traffic, completing the copy of the picture files as quickly as in the AVFS-cfq and *JFS-cfq* cases.

## 5. Related Work

Real-time disk scheduling has been extensively studied and many solutions proposed. One of the simplest solutions proposed is the Earliest Deadline First (EDF) policy which executes requests in increasing deadline order. It has been shown that EDF often leads to poor performance results compared to SCAN or C-SCAN due to a higher seek and rotational latency overhead [10]. SCAN-EDF [11], which also executes requests in increasing deadline order and increasing LBA order for requests with the same deadline, provides only marginal improvements over EDF. Both EDF and SCAN-EDF also lack a means to safely handle simultaneously best-effort requests while still guaranteeing the completion times of real-time requests.

This problem is addressed by more advanced scheduling methods such as RTFS [12] and ABISS [13]. However, unlike AVFS traffic mixer, these methods use fixed scheduling cycles for processing real-time requests which implicitly assumes periodic real-time behaviour of streams, i.e. a uniform increase in real-time request deadlines. It has been shown that the effectiveness of cycle based scheduling methods depends on the value chosen for the scheduling cycle [10]: short cycles can result in a high seek and rotational latency overhead, whereas larger periods degrade best-effort request processing response time. Since a DVR needs to support various video bit-rates at different playback speeds, determining an optimum processing cycle can be difficult in general cases and may require restrictions on the type of video and functions supported. Both RTFS and ABISS implementation also rely on stream buffers placed within the file system or kernel. This allows preserving the POSIX file I/O interface for processing real-time operations (using automatic read-ahead in the kernel buffers) but increases memory usage overhead due to the double buffering of data at the application and kernel levels if a traditional streaming application model is used. Overcoming this problem thus requires important design changes to existing applications. AVFS relies on a more traditional application model, making its integration in existing systems more simple. Also, its use of real-time requests deadlines and service time estimation leads to scheduling cycles that dynamically adapt to the workload.

The Cello scheduling framework [10] constitutes a more flexible approach to disk real-time scheduling. A class independent scheduler maintains a request execution queue according to request slack time (time until desired completion) determined by class specific schedulers. This results in efficient processing of mixed best-effort and real-time disk traffic, for instance by allowing fair-sharing of the disk bandwidth or disk processing time among service classes. However, there is no proposition for a real-time class specific scheduler beyond the low performing SCAN-EDF. The two level scheduling method mechanism of Cello allows the integration of the traffic mixer as a class specific scheduler, making these methods complementary.

Cooperative-I/O [14] utilizes a very different approach to providing QoS guarantees, albeit weaker ones. It introduces new I/O system calls to specify an acceptable delay for I/O requests which can in effect be seen as a deadline. Kernel level layers such as the virtual file system and the page cache are modified to use this information to delay the processing of disk I/O requests within the desired maximum delay, resulting in bursts of requests sent to the disk. This is in essence very comparable to AVFS but requires more changes to the operating system core parts, leading to a more difficult integration process and maintenance. AVFS has the advantage of being less intrusive as its benefits are confined within the file system and disk scheduler which are implemented as dynamically loadable modules.

Providing QoS guarantees for real-time applications also requires that the workload applied to the disk does not exceed its physical capabilities. This is usually implemented using bandwidth reservation methods. However, as disk performance heavily depends on head seek overhead and rotational latency generated by the sequence of requests executed, the maximum performance is not easily predictable. Often, the worst case (maximum seek latency and rotational delay between requests) is used for estimating maximum performance, leading to a non-optimal result [8]. Better estimates can be obtained through statistical methods but at the expense of weaker QoS guarantees [15]. However, unlike video streaming servers, the maximum number of streams that a DVR must process is often physically fixed by parameters such as the number of video tuners and the maximum number of clients that can access it. This results in a lower need for a bandwidth reservation function which can be replaced by a simple application

level stream admission control. A QoS measurement method such as introduced in Section 2 is a practical solution to precisely verify if a particular workload can be processed.

The data block allocation scheme used in the A/V file system is very similar to the region based block allocation (REBECA) mechanism [16]. This method also distributes data blocks evenly across the disk to lower on average seek distances between I/O operations and achieve more efficient processing of accesses to video files. The Symphony multimedia file system [15] also showed that a file system/disk scheduler integrated approach using a data block size adapted to the file type processed, as implemented by AVFS, significantly improves performance.

Previous work with AVFS has also shown that reducing seek overhead leads to significantly lower disk power consumption [17]. Compared to *cfq* and other Linux standard I/O schedulers, a reduction of up to 20% in power consumption can be observed.

## 6. Concluding Remarks

In this paper, we propose AVFS which is comprised of a file system and real-time disk scheduler, to comprehensively resolve problems in implementing real-time streaming applications in single disk embedded systems. The file system uses large data blocks evenly distributed over the disk partition to stabilize performance and to reduce the amount of metadata necessary to store file data block mapping on disk. The inclusion of deadline information to the proposed asynchronous I/O interface effectively allows lower level components in the storage stack to differentiate the type of file accesses (real-time or best-effort) while preserving a traditional buffer based application implementation model for stream processing. Support for AVFS at the application level can be achieved very easily for real-time applications without requiring any modifications whatsoever to other software. AVFS disk I/O scheduler, called the traffic mixer, processes real-time requests in batches built dynamically according to real-time request deadlines and service time estimation. This methods result in an important reduction of the average seek distance between consecutively executed requests and thus to better performance.

Evaluation results have shown that compared to traditional file systems and I/O schedulers, AVFS can maintain a high QoS level for real-time I/O operations even in the presence of best-effort disk accesses and under high workloads. Furthermore, AVFS performance is more stable with lower disk utilization rates, making it possible to use smaller and slower disk form factors (e.g. 2.5" disk drives) for processing high bit rate workloads such as high-definition video.

**Acknowledgments** The authors wish to express their sincere thanks to the following persons for their contribution to and support of this work: Dr. Richard New and Dr. Zvonimir Bandic of Hitachi Global Storage Technologies, San Jose Research Center, Mika Mizutani and Dr. Tadashi Takeuchi of Hitachi Ltd. Systems Development Laboratory and Prof. Hiroshi Nakashima of Kyoto University, Academic Center for Computing and Media Studies.

## References

1) De Nijs, G., Biesheuvel, A., Denissen, A. and Lambert, N.: The Effects of Filesystem Fragmentation, *Proc. 2006 Linux Symposium*, Vol.1, pp.193–208 (2006).
2) Wang, A.-I.A., Kuenning, G., Reiher, P. and Popek, G.: The Conquest file system: Better performance through a disk/persistent-RAM hybrid design, *Trans. Storage*, Vol.2, No.3, pp.309–348 (2006).
3) Seltzer, M., Chen, P. and Ousterhout, J.: Disk scheduling revisited, *Proc. USENIX Winter 1990 Technical Conference*, pp.313–324 (1990).
4) Jacob, B., Ng, S.W. and Wang, D.: *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann (2007). ISBN-13: 978-0123797513.
5) Axboe, J.: Time sliced cfq (2004). http://article.gmane.org/gmane.linux.kernel/264676
6) Tweedie, S.: Journaling the Linux ext2fs Filesystem, *Proc. 4th Annual Linux Expo* (1998).
7) Best, S.: JFS Overview (2000). http://jfs.sourceforge.net/project/pub/jfs.pdf
8) Mesut, O. and Lambert, N.: HDD Characterization for A/V Streaming Applications, *IEEE Transactions on Consumer Electronics*, Vol.48, No.2, pp.802–807 (2002).
9) Molaro, D., Payer, H. and Le Moal, D.: Tempo: Disk Drive Power Consumption Characterization and Modeling, *Proc. 13th IEEE International Symposium on Consumer Electronics* (*ISCE '09*), pp.246–250 (2009).
10) Shenoy, P. and Vin, H.M.: Cello: A Disk Scheduling Framework for Next Generation Operating Systems, *Proc. 1998 ACM SIGMETRICS Joint international Conference on Measurement and Modeling of Computer Systems*, pp.44–55 (1998).
11) Reddy, A.L.N., Wyllie, J. and Wijayaratne, K.B.R.: Disk scheduling in a multimedia I/O system, *ACM Trans. Multimedia Comput. Commun. Appl.*, Vol.1, No.1,

pp.37–59 (2005).
12) Hong, L., Cumpson, S., Jochemsen, R., Korst, J. and Lambert, N.: A scalable HDD video recording solution using a real-time file system, *IEEE Transactions on Consumer Electronics*, Vol.49, No.3, pp.663–669 (2003).
13) De Nijs, G., Van Den Brink, B. and Almesberger, W.: Active Block I/O Scheduling System (ABISS), *Proc. 2005 Linux Symposium*, Vol.1, pp.109–126 (2005).
14) Weissel, A., Beutel, B. and Bellosa, F.: Cooperative I/O: a novel I/O semantics for energy-aware applications, *OSDI '02: Proc. 5th symposium on Operating systems design and implementation*, pp.117–129 (2002).
15) Shenoy, P.J., Goyal, P., Rao, S. and Vin, H.M.: Symphony: an Integrated Multimedia File System, *Technical Report* (1998). UMI Order Number: CS-TR-97-09.
16) Ghandeharizadeh, S., Kim, S.H. and Shalabi, C.: On configuring a single disk continuous media server, *Proc. 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.37–46 (1995).
17) Le Moal, D., Molaro, D. and Campello, J.: Power efficient real-time disk scheduling, *NOSSDAV '09: Proc. 18th international workshop on Network and operating systems support for digital audio and video*, pp.55–60 (2009).

(Received July 24, 2009)
(Accepted December 9, 2009)

**Damien Le Moal** received his engineering degree in Computer Science and Applied Mathematics from ENSEEIHT (National Superior Institute of Electronics, Electrical Engineering, Computer Science, Hydraulics and Telecommunications, Toulouse, France) in 1995 and his M.I. from Kyoto University, Graduate School of Informatics, in 2000. Since joining Hitachi Ltd. in 2000, he has been engaged in research on operating systems, storage systems and applications for video streaming.

**Donald Molaro** is a Sr. Software Engineer with Hitachi Global Storage Technologies, San Jose Research Center, holds a MSc. from the University of Calgary, Canada, and has over twenty years of software development experience. He has worked on a number of consumer and professional electronic products including set-top and media server systems. Since joining Hitachi Global Storage Technologies in August 2004, he has worked on addressing several issues related to the use of hard-disks in high-performance multimedia systems.

**Jorge Campello** received EE and M.Sc. in Electrical Engineering degrees from Universidade Federal de Pernambuco, Recife, Brazil in 1992 and 1994 respectively. He received a Ph.D. in Electrical Engineering from Stanford University, Palo Alto, California, in 1999. In 1999 he joined IBM's Almaden Research Center as a Research Staff Member, working on Coding and Information Theory applied to Magnetic Recording Systems. In 2003, he joined Hitachi Global Storage Technologies San Jose Research Center where he is currently a research staff member working in the area of hard-disk drive applications to Consumer Electronics.