Apollon : File System Level Support for QoS Augmented I/O

Taeseok Kim[1], Youjip Won[2], Doohan Kim[2], Kern Koh[1], and Yong H. Shin[3]

[1] School of Computer Science and Engineering, Seoul National University,
56-1, Shillim-Dong, Kwanak-Ku, Seoul, 151-742, Korea
{tskim, kernkoh}@oslab.snu.ac.kr
[2] Division of Electrical and Computer Engineering, Hanyang University,
17, Hangdang-Dong, Seongdong-Ku, Seoul, 133-791, Korea
{yjwon, lissom33}@ece.hanyang.ac.kr
[3] Dept. of Computer Science and Engineering, Seoul National University of Technology,
172, Gongreung-Dong, Nowon-Ku, Seoul, 139-743, Korea
yshin@snut.ac.kr

Abstract. Next generation information appliances are required to handle real-time audio/video playback and in the mean time should be able to handle text based requests such as database search, file recording, etc. Although several techniques are presented to address this problem, most of them are rather theoretical to be employed into practical systems as they are. In this paper, we present our experience in developing the file system which can efficiently handle mixed workload. To this end, we develop practical I/O scheduling mechanism to prioritize the incoming disk I/O requests: deadline-driven I/O scheduler and admission control module. We also discuss some issues on QoS enhanced I/O semantics. The proto-type file system Apollon is developed on Linux Operating System. Compared to legacy system, Apollon exhibits superior performance in guaranteeing the QoS requirement of real-time requests.

1 Introduction

With the penetration of computer technologies into consumer electronics platform, the usage of digital home appliances such as digital TV, set-top box and PVR(Personalized Video Recorder) have explosively increased. These embedded devices enable us to enjoy service such as interactive multimedia presentation without the full fledged computer system, e.g. desktop, laptop computer, etc. They are usually equipped with relatively limited computing capability: smaller amount of main memory and storage. They also have to fulfill new line of constraints which have been applied to consumer electronics platform: reliability, shock-resistance, and energy consumption, etc. Legacy operating systems carry too much weight to be used in these embedded devices. Hence, it is mandatory that the operating system for these embedded devices is carefully tailored to satisfy the specific constraints of the devices.

In this work, we focus our effort on developing file system which can efficiently handle audio/video workload as well as the I/O workload without timing constraints. Developing this file system is motivated by the actual need. ATSC standard requires 19.2 Mbits/sec playback rate[1]. And single set-top box or PVR device is required to handle at least two read and two write sessions of ATSC stream. This type of real-time requirement has not existed in legacy computer systems domain, it should be necessarily considered in designing the I/O subsystem for multimedia embedded devices. In addition, the device needs to handle aperiodic I/O request, e.g. file download, database search, etc. This requirement will be more realistic when TV-anytime or infomercial is combined with real-time database navigation capability.

Due to the head movement overhead of disk and the stringent real-time characteristics of video/audio workload in home appliances, the file system in such embedded devices requires rather sophisticated treatment. The easiest way to overcome this situation is to allocate different device[1] to each type of data. However, this approach cannot use the underlying resources efficiently[2]. In this work, we consider that single device is required to provide streaming service as well as to handle the requests for non-playback related data.

In fact, guaranteeing QoS under mixed workload has been under serious attention for the past few years[3-5]. Most of the techniques including Cello[3] employ period based scheme. Period based approach delivers very sophisticated model and can exploit that underlying resources efficiently. However, it mandates the in-depth knowledge of the disk internals, e.g. the number of cylinders, sectors/cylinder, seek distance vs. seek time curve which are not usually accessible from operating system's point of view. Further worse, it requires more CPU cycles to schedule the request. Another common approach for servicing the mixed I/O requests is to employ a scheduler that assigns priorities to application classes and services disk requests in the priority order[6-7]. Unfortunately, such scheduler may violate service requirements of lower priority requests and induce long-time starvation.

In this paper, we present simple yet efficient method of handling different I/O requests. We first classify all I/O requests into two categories: real-time requests and best-effort requests. And then we develop a deadline-driven I/O scheduler which can not only meet the deadlines of real-time requests but also offer good response time to best-effort requests. For jitter-free service for audio/video under bursty workload environment, we also supplement our scheduling scheme with admission control module. Finally, we discuss some issues on QoS enhanced I/O semantics in commodity operating system. We demonstrate that Apollon is suitable for next generation embedded devices since: (i) it provides the QoS service for audio/video playback requests, (ii) it is so simple to be employed into low capability devices, and (iii) it has practical assumptions to be implemented as it is.

The rest of this paper is organized as follows. First, Apollon architecture and Apollon APIs are presented in Section 2, Section 3, respectively. And then, we discuss the prototype implementation in Section 4 and show the efficacy of Apollon through extensive experiment in Section 5. Finally, we conclude the paper in Section 6.

[1] The device denotes the physical device which is a separate scheduling entity.

2 Apollon File System

2.1 Apollon File System Architecture

Apollon file system is designed for embedded device with limited computing capability, e.g. a few hundred MHz at most. It naturally raises two main design philosophy: small code size and less CPU overhead. Apollon is very simple, and yet still successfully guarantees the required bandwidth for multimedia stream. Fig.1 illustrates the architecture of Apollon. We classify all I/O requests into two categories; soft real-time I/O, e.g. I/O request for audio/video playback and best-effort I/O, e.g. file copy, file transfer, or I/O request originated from database search. As is the case for all real-time operations, real-time I/O request should accompany deadline requirement. However, in POSIX standard, file system interfaces do not carry deadline information. Hence, in Apollon file system, we augment I/O system calls with deadline to support real-time I/O. Details on specifying QoS of real-time requests will be discussed in Section 3. System call can also be called without deadline such as best-effort I/O request. When the system call is called without deadline, file system layer assigns infinite value for deadline field. Based this deadline information, we implemented deadline-driven I/O scheduler: Earliest Deadline First with Aging. We also added admission control module to prevent the system from being overloaded. We describe admission control module and deadline-driven I/O scheduler one after another in next subsections.

2.2 Admission Control Module for Apollon

To service a given I/O request satisfying its deadline requirement, the system should not be overloaded. There are a number of levels in determining whether to accept a given I/O request or not: (i) open/close level, (ii) session level, and (iii) I/O request level. When we control the admission in open/close level, the application specifies the deadline requirement(or playback rate of a file) when file is open. The file system computes the resource requirements of a given open system call and rejects that if it cannot guarantee a given deadline. The second approach is at session level. Session is

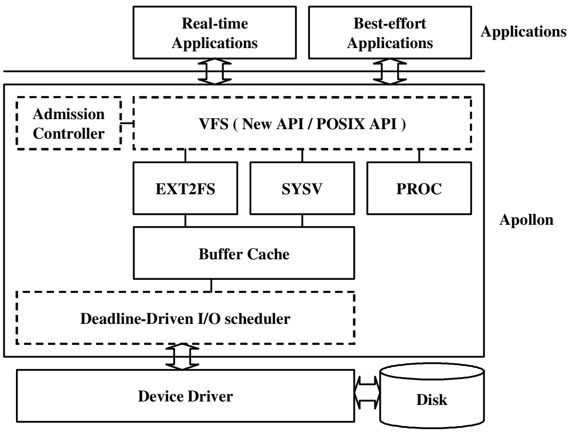


Fig. 1. Architecture of Apollon file system. The dotted line boxes denote the modules which are implemented by Apollon.

a sequence of homogeneous QoS I/O requests. In VCR like operation, user can watch the video in a number of different playback speeds. The admission control module recomputes the resource requirement of the playback when the user updates the playback speed and determines whether to accept the given session or not. Finally, we can determine the admissibility of a request for individual I/O request. In Apollon, we adopt the first approach.

The continuity requirement to maintain jitter-free playback can be represented with two conditions. The first condition is that the number of data blocks retrieved during time *T* should be greater than the amount of data blocks needed for playback for same period of time. This condition could be denoted by Eq.(1).

$$T \cdot r_i < n_i \cdot b \tag{1}$$

In Eq.(1), n_i, b, and r_i are the number of data blocks read during *T*, size of I/O unit, and playback rate of stream *i*, respectively. This condition should hold for each stream, $i = 1,\ldots, m$. The second condition is that it should take less than time *T* to retrieve the data blocks for all streams.

$$T \geq \{\sum_{i=1}^{m} \frac{n_i \cdot b}{B_{\max}}\} + O(m) \tag{2}$$

In Eq.(2), *m* and B_{max} are the number of streams, maximum transfer rate of the disk, respectively. And *O(m)* is the disk movement overhead such as seek and rotation latency in reading the data blocks for *m* streams. It is important to note that *O(m)* is determined by the disk scheduling policy, e.g. EDF, SCAN, FIFO, etc. When the player opens a multimedia file, admission control module is required to compute above two equations and determines whether to accept or reject the I/O requests.

2.3 Deadline-Driven I/O Scheduler: Earliest Deadline First with Aging

Based deadline and data location on disk, we develop novel scheduling strategy. In this scheduling strategy, each request has deadline, location of a requested block on disk, and age. Our I/O scheduler first services the request with the earliest deadline. Since real-time requests have specific deadline value while best-effort requests have infinite value, real-time requests are first served. If the two or more requests have the same deadline, they are serviced in increasing sector number order.

Since best-effort I/O requests are just prioritized by data location on disk, it is possible that best-effort I/O request is indefinitely postponed. We adopt the notion of “age” to overcome this situation. The notion of aging is being widely used in commodity operating system such as Linux 2.4. Each request in the queue has an age and it is initialized to 0. The age of a request increases whenever other requests which have same deadline but lower sector number pass that request. If the age of a request comes to a threshold τ, any request is not allowed to pass that request. This threshold value τ means the maximum latency of the request. For example, when new request R_k with lower sector number than a request R_i arrives at queue, if R_i has a timed out age, R_k cannot pass R_i. Consequently, a request with a timed out age acts as a barrier and this mechanism resolves the starvation problem. Pseudo code of our algorithm is explained in table 1.

Table 1. Pseudo Code of Deadline-driven I/O Scheduler

```
$R_i$ ($d_i$, $s_i$, $a_i$)
$d_i$ : deadline of request $i$
$s_i$ : sector number of request $i$
$a_i$ : age of request $i$ for preventing long starvation

when new request $R_k$ arrives at queue
for ( i := N; i >0; i--) do
    if ( $d_k > d_i$ ) then insert $R_k$ after $R_i$ and return;
    else if ( $d_k = d_i$ ) then
        if ( $s_k > s_i$ | $a_i$ = threshold ) then insert $R_k$ after $R_i$ and return;
        else $a_i$ ++;
        end if;
    end if;
end for;
insert $R_k$ at head of queue;
```

2.4 Analysis of Deadline-Driven I/O Scheduling Algorithm

In this section, we discuss the performance analysis of our scheduling algorithm. $T_{service}$, the time required to service N requests in queue is modeled as Eq.(3). In Eq.(3), B_{max} is the maximum data transfer rate which is governed by the rotational speed and magnetic density of the disk plate. n_i, b, N represent the number of blocks to be fetched for *i*th request operation , the block size and the number of requests in queue, respectively. And δ_i in Eq.(3) is the disk head repositioning overhead for *i*th request I/O operation.

$$\begin{aligned} \mathrm{T}_{\mathrm{service}} &= \sum_{i=1}^{N} \left(\frac{bn_i}{B_{\max}} + \delta_i\right) \\ &= \sum_{i=1}^{N} \frac{bn_i}{B_{\max}} + \sum_{i=1}^{N} \delta_i \end{aligned} \tag{3}$$

We can partition Eq.(3) into two parts, namely, the data transfer time, which is $\sum_{i=1}^{N} \frac{bn_i}{B_{\max}}$, and the positioning overhead, $\sum_{i=1}^{N} \delta_i$. The data transfer time depends on the total number of blocks to be transferred, while the positioning time is governed by the disk scheduling policy. The positioning time consists of seek time, rotation latency, settle down time, head change, etc. Among all of them, seek operation dominates the disk head positioning time and thus it should be modeled carefully.

We first model the seek time behavior of the disk head movement. A number of experimental measurements have resulted in the following model of seek time behavior. Here, x and C denote the seek distance and threshold value, respectively, in terms of the number of cylinders.

$$\begin{aligned} T_{seek} &= a_1 + b_1\sqrt{x}, \quad \text{if} \quad x \le C \\ T_{seek} &= a_2 + b_2 x, \quad \text{if} \quad x > C \end{aligned} \tag{4}$$

In FIFO or EDF disk scheduling algorithm, the disk heads read/write the data blocks in a fixed order independent of the location of the data blocks. Due to this property of FIFO or EDF scheduling, the disk head makes a full sweep of the disk platter (N -1) times in the worst case. The corresponding maximum overhead in FIFO or EDF scheduling, $O_{FIFO/EDF}(seek)$ is expressed as in Eq.(5). In Eq.(5), L means the total number of disk cylinders.

$$O_{FIFO/EDF}(seek) = (N-1)(a_2 + b_2 L) \tag{5}$$

In SCAN scheduling algorithm, the disk head scans the cylinder from the center of the platter outwards (or vice versa), and reads the data blocks in cylinder order. Thus, the respective cylinders are visited once in each cycle. The upper bound on the overhead in SCAN algorithm, $O_{SCAN}(seek)$ is expressed as in Eq.(6), given that $\frac{L}{N-1} \le C$. This formulation is due to the fact that when the inter-cylinder distance is shorter than a certain threshold, seek time is proportional to the square root of the distance.

$$O_{SCAN}(seek) = (N-1)(a_1 + b_1\sqrt{\frac{L}{N-1}}) \tag{6}$$

In SCAN with aging, the disk head scans the cylinders like SCAN algorithm except prior service for requests with timed out age. In this algorithm, due to aging threshold value τ, the set of requests to be serviced in worst case are partitioned into $\left\lceil \frac{N}{\tau+1} \right\rceil$ groups. SCAN scheduling is used within a group and FIFO scheduling is used between the groups. Since there are $\left\lceil \frac{N}{\tau+1} \right\rceil$ groups to be scanned, the disk head makes $\left\lceil \frac{N}{\tau+1} \right\rceil$ sweeps of the disk platter. With this figure, the positioning overhead in SCAN with aging, $O_{SCANA}(seek)$ can be formulated as in Eq.(7). In Eq.(7), when τ is equal to (N-1), $O_{SCANA}(seek)$ becomes the same as $O_{SCAN}(seek)$ and when τ is equal to 0, $O_{SCANA}(seek)$ becomes the same as $O_{FIFO}(seek)$. Given that $\frac{L}{\tau} \le C$, $O_{SCANA}(seek)$ is expressed as in Eq.(7).

$$O_{SCANA}(seek) = (N - \left\lceil \frac{N}{\tau+1} \right\rceil)(a_1 + b_1\sqrt{\frac{L}{\tau}}) + (\left\lceil \frac{N}{\tau+1} \right\rceil - 1)(a_2 + b_2 L) \tag{7}$$

From above equations, we can derive worst case seek time overhead in our deadline-driven I/O scheduling: Earliest Deadline First with aging. In this algorithm, real-time requests are serviced in EDF order while best-effort requests are serviced in SCAN with aging algorithm order. Let N_r and N_b be the number of real-time requests and the number of best-effort requests in queue, respectively. The disk head makes a full sweep of the disk platter N_r times and then scans the cylinders like SCAN with aging algorithm for N_b best-effort requests. Given that $\frac{L}{\tau} \le C$, $O_{EDFA}(seek)$ is expressed as in Eq.(8).

$$\begin{aligned} O_{EDFA}(seek) &= (N_r - 1)(a_2 + b_2 L) + (a_2 + b_2 L) \\ &+ (N_b - \left\lceil \frac{N_b}{\tau+1} \right\rceil)(a_1 + b_1\sqrt{\frac{L}{\tau}}) + (\left\lceil \frac{N_b}{\tau+1} \right\rceil - 1)(a_2 + b_2 L) \end{aligned} \tag{8}$$

3 QoS Enhanced I/O Semantics

3.1 QoS Semantics in I/O Operation

To support QoS in I/O subsystem, operating system should harbor proper abstraction of QoS in I/O operation. In fact, the notion of QoS is unpopular in I/O operation. Traditionally, hard disk has been outside the realm of real-time computing. This is because seek time and rotational latency have made it infeasible to guarantee response time in hard real-time environment. I/O subsystem which is based on such disk is not friendly to real-time system and there is no consideration for specifying timing constraints in POSIX. Hence, to support QoS of real-time requests in I/O subsystem, we first have to define abstract mechanism for specifying QoS.

There are several alternatives in specifying QoS requirement of multimedia applications. The simplest way is to add new APIs including QoS related parameter such as deadline. In [8], they implemented new system calls such as `cello_open()`, `cello_read()`, etc. and added QoS parameter into those APIs. [9] specifies QoS parameters per file descriptor, and passes the pointer to parameters to each IO request in the disk queue using `ioctl()`.

The mechanism which classifies the requests class by using file expansion, e.g. `mpeg`, `avi`, etc. deserves much consideration. In other words, when opening a file, if its expansion name is one of audio/video file expansions, it is possible to prioritize the real-time requests in I/O level. In this scenario, however, the process which opens audio/video files also should be considered with file expansion. It is because, non-multimedia processes such as *copy* also could open the audio/video files. These processes do not require real-time services. It is also possible to classify the file characteristics in I/O level by analyzing the I/O patterns of files. For example, if the access pattern of a file is sequential and periodic, it could be recognized as stream requests. For this autonomous detection of requests class, the intelligent module for analyzing the file access pattern should be embedded in file system.

3.2 Interface in Apollon

Among several techniques, we added new APIs in order to pass the deadline information of multimedia requests into I/O level. The addition of new system calls causes application which is supposed to use new system calls to be changed. In general purpose system, since there exist many kinds of multimedia applications and they are frequently changed, it is not feasible. On the other hand, in embedded devices such as PVR or set-top box, applications are embedded along with operating system, and thus the above mentioned problems do not matter. Note that the addition of new system calls should be carefully handled because system call index is one of system resources. Though Apollon is yet prototype and thus this simple technique is employed, it might be replaced with other techniques such as QoS parameter passing with `ioctl()`. Table 2 lists the interface exported by Apollon.

Table 2. The Description of Apollon APIs

System Call	Purpose
apollon_open	invoke admission controller and if admitted, open the multimedia file
apollon_close	close the opened multimedia file and reset the admission control module related parameters
apollon_read	read multimedia file with its own deadline parameter
apollon_admin	set and modify the admission control related parameters

4 Implementation of Apollon Prototype

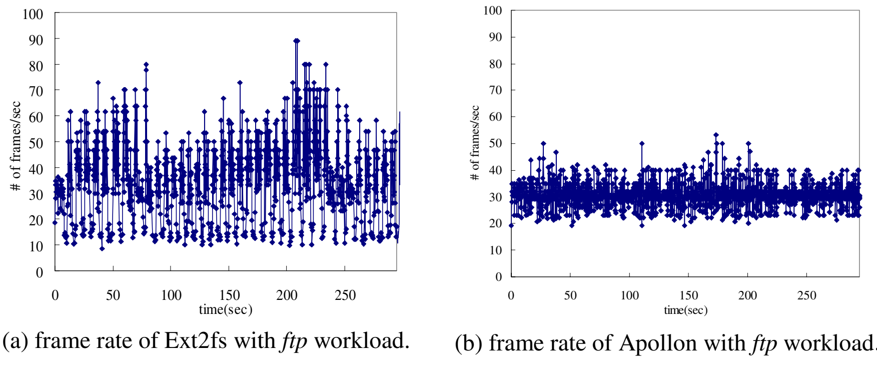
We implemented Apollon file system on Linux kernel v.2.4.20. Since Apollon file system enables the application to specify the deadline of I/O request, several kernel components are re-designed to harbor multimedia related information such as deadline, average playback rate. We also replaced Linux elevator disk scheduler with our deadline-driven I/O scheduler. Note that Apollon file system is completely modular and object-oriented. Hence, it can be seamlessly integrated with the existing file systems, e.g. Ext2fs, XFS, NTFS, etc.

5 Experimental Evaluation of Apollon

5.1 Experimental Methodology

Since Apollon exports QoS-enabled APIs, we modified the commodity mpeg2 player to use this feature[10]. The modified player computes the deadline for every I/O request and invokes the system call with that deadline. The objective of Apollon file system is to guarantee deadline of real-time I/O requests so that multimedia player can provide jitter-free playback. When the I/O subsystem is under utilized, it is not much difficult to deliver the requested data block on time. However, when the I/O subsystem is overloaded with various type of request, special care needs to be taken to guarantee the QoS of real-time I/O.

To evaluate the efficacy of Apollon file system, we compare the deadline guarantee behavior of Apollon with that of legacy Ext2fs using SCAN-like disk scheduler. Although several techniques are presented to support QoS enabled I/O, most of them are not fully implemented and thus they could not be used in our experiment. Instead, we show the performance overhead as well as the deadline guarantee behavior of our Apollon using intensive comparison with Ext2fs. To generate various types of request, we use *ftp*, *find* and *IOZONE*. *Ftp*(file transfer protocol) receives the data blocks from TCP socket and copies it to the disk. *Find* searches the directory entry to find the path of a given file. And *IOZONE* is a file system benchmark tool which measures the performance of a given file system[11]. Multimedia file used in this test is 9 Mbits/sec with 30 frames/sec playback rate. The size of the file is 324 MByte. And the testbed for experiments consists of a 2.4GHz Intel Pentium Ⅳ machine running Linux 2.4.20, equipped with 256MB RAM and a 40GB IDE disk.



(a) frame rate of Ext2fs with *ftp* workload. (b) frame rate of Apollon with *ftp* workload.

Fig. 2. Variation in playback frame rate when using *ftp* for background task

5.2 QoS Guarantee Under I/O Intensive Background Workload

We first use *ftp* for I/O intensive background task. Experiment with *ftp* could be regarded as simulation of recording another program during watching a movie in PVR. Fig.2 illustrates the variation of playback rates in both file systems when the *ftp*

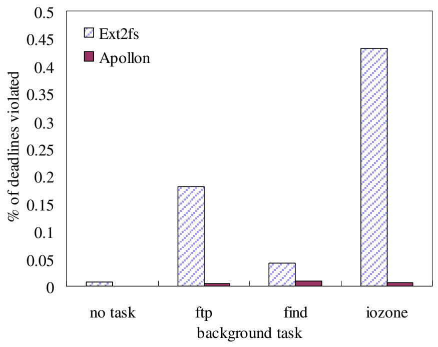


Fig. 3. Deadline violation behavior in different environment. In every case, deadline miss rate in Apollon file system is near zero.

application downloads the 2GB sized file. When we watch the video clip from Ext2fs with *ftp*, the playback speed fluctuates widely and thus it is actually impossible to watch the video clip(Fig.2(a)). However, the variation of playback rate in Apollon is relatively consistent(Fig.2(b)).

Fig.3 illustrates the deadline violation behavior in two file systems using different workload. In Fig.3, Y axis denotes the percentage of I/O requests which do not meet the deadline requirement. For this experiment, we played a movie file when there is no background task, *ftp* application downloads the 2GB sized file, *find* program searches the root file system, and *iozone* benchmark tool is executed with '-a' option, respectively. As can be seen, in every case, the deadline miss rate is near zero in Apollon file system. On the other hand, the deadline miss rate could not be negligible in Ext2fs file system.

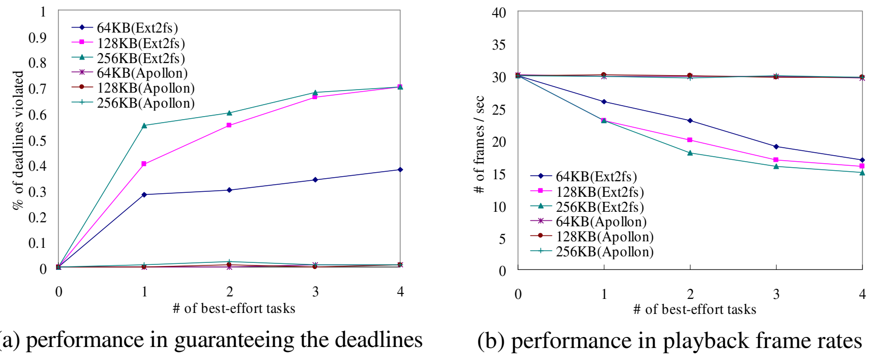
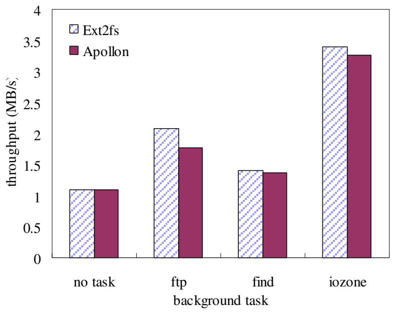


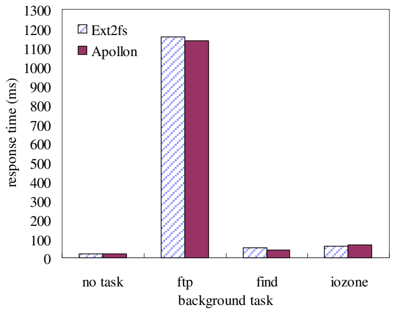
Fig. 4. QoS guarantee behavior of real-time I/O requests using *IOZONE*. Note that three lines in Apollon file system are overlapped one another. Although the performance is different as I/O size of multimedia player, Apollon exhibits superior performance.

Next, we use *IOZONE* benchmark tool to compare the performance of two file systems under extreme text based workloads. In Fig.4(a), X axis denotes the number of *IOZONE* tasks and Y axis is the percentage of I/O requests which do not meet the deadline requirement. In this figure, one unit of *IOZONE* task means an *IOZONE* session that reads and writes 128MB sized file by 4KB. As can be seen, Apollon file system exhibits superior performance in guaranteeing the deadline of I/O requests. Fig.4(b) plots the average playback frame rates in two file systems. In Ext2fs, the playback rate of player decreases as we increase the number of *IOZONE* tasks. However, in Apollon file system, the playback rate of the player remains constant independent of the number of *IOZONE* background tasks.

Compared with Ext2fs using seek optimized scheduling like SCAN, Apollon file system involves the overhead in disk head moving optimization due to the QoS service for real-time I/O requests. To show the overhead of Apollon, we measure the throughput and the average response time of requests. These measurements are taken under identical conditions with Fig.3. As shown in Fig.5(a), Apollon incurs more overhead than Ext2fs by 3% ~ 14% in throughput. Fig.5(b) also shows that there is little difference in average response time between two file systems. Although Ext2fs is a little better than Apollon in seek optimization performance, it dose not guarantee meeting the deadline requirements of the real-time requests. We have recorded the playback in Apollon file system and Ext2fs. Interested users are referred in [12].



(a) performance in throughput



(b) performance in average response time

Fig. 5. The throughput and the average response time in two file systems

6 Conclusion

In this paper, we presented the integrated file system for handling mixed workload. Using admission controller and deadline-driven I/O disk scheduler which prioritizes the different types of requests according to their respective deadline requirements, we were able to successfully meet the soft real-time requirement of audio/video application under mixed workload. We also discussed the limit of POSIX in supporting the real-time characteristics in I/O level and then presented several solutions to relieve that conservative semantics. In the work presented here, we added new system calls for prototype file system, we are currently investigating issues in classifying efficiently the requests class by file expansion or autonomous detection. Apollon file system manifests itself especially when the given system is equipped with relatively low end disk subsystem and the system is required to exploit its capacity. This file system is to be embedded in the digital home appliances, e.g. set-top box, PVR, internet home server, etc.

Acknowledgement

This work is in part funded by KOSEF through Statistical Research Center for Complex System at Seoul National University.

References

1. http://www.atsc.org/standards.html
2. P. Shenoy, P. Goyal, and H. Vin: Architectural considerations for next generation file systems. Proceedings of ACM Multimedia Conference, Orlando, FL, USA (1999) 457-467
3. P. Shenoy: Cello: a disk scheduling framework for next generation operating system. Real Time Systems Journal (2002)
4. R. Wijayaratne and A. L. Reddy: Providing QoS guarantees or disk I/O. Proceedings of ACM/Springer Journal on Multimedia Systems (2000)
5. Y. J. Won and Y. S. Ryu: Handling sporadic tasks in multimedia file system. Proceedings of the eighth ACM International Conference on Multimedia (2000) 462-464
6. A. L. Reddy and J. Wyllie: Disk scheduling in multimedia I/O system. Proceedings of ACM Multimedia'93, Anaheim, CA (1993) 225-234
7. K. Gopalan: Real-time disk scheduling using deadline sensitive SCAN. Technical Report TR-92, Experimental Computer Systems Labs, Dept. of Computer Science, State University of New York, Stony Brook (2001)
8. V. Sundaram, A. Chandra, P. Goyal, P.Shenoy, J. Sahni, and H. Vin: Application performance in the QLinux multimedia operating system. Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA (2000) 127-136
9. Z. Dimitrijevic and R. Rangaswami: Quality of service support for real-time storage systems. Proceedings of International IPSI-2003 Conference, St. Stefan, Montenegro (2003)
10. http://libmpeg2.sourceforge.net
11. http://www.iozone.org
12. http://www.dmclab.hanyang.ac.kr/research/project/hermes-q/hermes-q_overview.htm