ON-THE-FLY TIME SCALING FOR COMPRESSED AUDIO STREAMS

Suzana Maranhão, Rogério Rodrigues and Luiz Soares
166 166 TeleMidia Lab, PUC-Rio University
Brazil, Rio de Janeiro

Keywords: Time scaling; High-Quality Compressed Audio; MPEG-2 Audio; MPEG-4 Audio, MPEG-2 Systems; Inter-media Synchronization; Hypermedia Document Presentation.

Abstract: Time scaling is a technique used to modify media-object presentation duration. This paper proposes an audio time-scaling algorithm focused on supporting applications that need: to maintain the original data format for storage or immediate presentation on any legacy audio player; to perform linear time scaling in real time, allowing the adjustment factor to vary along the audio presentation; and to perform time mark-up maintenance, that is, to compute new time values for original marked audio time instants. The proposed algorithm is appropriate for those applications that do not need a great adjustment factor variation. The integration with content rendering tools is presented in the paper and also an example of using these tools in a hypermedia presentation formatter..

1 INTRODUCTION

Time scaling and elastic time adjustment are usual names given to the technique of modifying media-object playing duration. The adjustment can be quantified by a (tuning or adjustment) factor, expressed by a real number greater than zero. A media exhibition speed up is expressed by a tuning factor $f<1$ and a media exhibition slow down by a factor $f>1$.

Time-scale modifications are useful in many applications (Aron, 1992; Lee, 2004; Omoigui, 1999). The next paragraphs give some examples that raised the goals of the algorithm proposed in this paper. As will become evident, hypermedia (or interactive multimedia) systems are our main target, reason why the proposed algorithm is called *HyperAudioScaling*.

The first example comes from the temporal synchronization consistency of hypermedia document presentations. In such presentations, all media-objects should be exhibited in proper times taking into account time restrictions specified by authors, including interactive actions. Time-scaling processing should be applied before document presentation (at compile time) trying to solve all temporal inconsistencies. However, the temporal synchronized media-objects can lose their synchronization due to several factors in their route to the presentation machine (from now on called *formatter*). For example, network delays, user interactions and unpredictable end of media exhibitions (as in live streams) can cause synchronization loss. In all theses cases, time scaling will play an important role in the synchronization maintenance and must be performed in presentation time (on-the-fly). Moreover, to track the synchronization, the adjustment factor may also vary in real time. The fidelity of the processed media must be high and, thus, adjustments must be performed as smoothly as possible, in order to be imperceptible to users.

The adjustment mechanism should not depend on which audio presentation tool (i.e. audio player) will be used by a hypermedia formatter. As a consequence, time-scale modifications must be independent of the stream decoding process that takes place in presentation tools. Moreover, since hypermedia applications usually manipulate compressed media formats, and since real-time time-scale modifications are needed, the adjustments should be performed in the compressed data, without needing to decode them.

Finally, hypermedia documents use anchors (defined in this paper as time periods in the stream) to specify synchronization points. So, the time-scaling algorithm should also be able to track new anchor values during the adjustment computation.

Maranhão S., Rodrigues R. and Soares L. (2006).
ON-THE-FLY TIME SCALING FOR COMPRESSED AUDIO STREAMS.
In *Proceedings of the International Conference on Signal Processing and Multimedia Applications*, pages 110-117
DOI: 10.5220/0001570901100117

Other multimedia applications also demand time-scaling processing. For example, time scaling can be useful to allow TV or radio stations to speed its scheduling up or down. Time scaling can also be used to optimize channel allocation for multiple users in a Video (or audio)-On-Demand (VOD) system. Media streams of the same content can be transmitted with different speeds until the same piece of information is reached in the flows. At this time, the streams can be unified in one multicast flow.

The HyperAudioScaling algorithm, proposed in this paper, focuses on supporting audio time scaling for those applications that need: i) to maintain the original data format aiming at the storage or immediate presentation on any legacy audio player; ii) to perform time-scale modifications in real time (presentation time) but allowing to vary the adjustment factor during the audio presentation; iii) high fidelity, that is, given an adjustment factor, the time-scaling processing should be performed as smoothly as possible in order that speed variation in the resultant media is imperceptible to users; and iv) time mark-up maintenance, that is, time scaling must be able to compute new time values for original marked time instants. In addition, the HyperAudioScaling algorithm focuses only on those applications that do not need a great tuning-factor variation ($0.9 \leq f \leq 1.1$).

This paper is organized as follows. Section 2 discusses some related work. Section 3 introduces the HyperAudioScaling algorithm for audio-only streams. Section 4 briefly discusses how the algorithm can also be applied in multiplexed audio and video (system) streams. Section 5 presents the time-scaling tool (library) developed based on the proposed algorithm. In order to illustrate the use of the time-scaling tool in a hypermedia document presentation system, Section 6 describes its integration with a hypermedia formatter. To conclude, Section 7 presents the final remarks.

2 RELATED WORK

Audio time-scaling algorithms can be classified in three categories (Bonada, 2002; Lee, 2004), presented in increasing order of quality and computational complexity.

Time-based algorithms assume signal segmentation in time domain in order to perform adjustments by segment manipulations (for example, through discarding, duplication or interpolation of segments). They usually reach good quality for adjustment factor between *0.8* and *1.2*.

Frequency-based algorithms perform time scaling, modifying frequencies of the original audio signal. They can produce high quality output over a wide range of stretching factors.

Analysis-based algorithms make an examination of the audio signal and then perform time-scaling mechanisms specifics for that audio type, in order to create an adjusted high-quality audio.

Most of the commercial tools use time or frequency-based algorithms for adjustments on generic audio, once the computational complexity of algorithms based on detailed analysis is usually too high.

Independent of the time-scaling algorithm category, there are, at least, tree ways to perform adjustments in compressed audio. The first possibility is to decode, process, and recode the stream. The advantage of this solution is that there are many good time-scaling algorithms proposed for uncompressed audio (Lee, 2004). However, this option can be very time-consuming, making it difficult to be used in real-time. Furthermore, there is a loss of quality associated with the recompression process.

When audio streams must be processed while they keep playing, originating new streams without perceptible delay, there are two other different solutions to perform time-scale modifications.

The first one is carrying out time scaling soon after the decompression and before the presentation. The advantage in this case is the possibility of manipulating non-compressed streams using the aforementioned good algorithms already defined in the literature. However, it may be difficult to intercept the decoder output before sending it to exhibition. Even when this interception is possible, it may require a particular implementation for each decoder.

The second solution is performing time scaling before decompression, straight on compressed streams. This option gives to the time-scaling algorithm independency from the decoder. However, time scaling must consider the syntax rules specified by the media format. Note that, for the requirements stated in Section 1, this is the approach to be followed.

Sound Forge (Sony, 2006) allows professional audio recording and editing. It supports several audio formats, but always applies a pre-processing procedure when opening compressed files. Time-scaling processing can be performed in presentation time, however, probably using the pre-processed stream. Nevertheless, the generation of the resultant (recompressed) file is not performed in real time. The user can choose among *19* different ways of applying time-scale modifications according to his/her needs. The adjustment factor varies between

0.5 and *5*. The tool has an excellent audio output stream quality, but introduces an intolerable delay for real time processing. Sound Forge is used for comparison with this paper proposal in Section 7.

Windows Media Player 10 (Microsoft, 2006) supports many audiovisual formats, such as MP3 and WMA. The tool allows choosing the adjustment factor to be used during media exhibition and to change this factor in presentation time. Although the adjustment factor can assume values between *0.06* and *16*, the program specification suggests a range between *0.5* and *2.0* to keep media quality high. Although not mentioned, it is almost assured that the time-scaling processing takes place after the decoding, due to two reasons: it is not possible to save the processed audio in a compressed format; and, since the algorithm is done for a specific player, it is better to apply time scaling just before presentation.

The MPEG-4 audio specification defines a presentation tool called PICOLA (Pointer Interval Controlled OverLap Add), which can make time-based adjustments after decompression, in mono audio with sample rate of *8*kHz or *16*kHz (ISO, 2001).

FastMPEG (Covell, 2001) is a time-scaling proposal that explores the partial decoding/encoding strategy. It describes three time-based algorithms for MP2 format on-the-fly adjustments. The algorithms are performed after a partial decoding of the audio stream and followed by a partial re-encoding. The adjustment factor varies between *2/3* and *2.0*.

All aforementioned time-scaling algorithms are not applied straight on the compressed stream. Instead, streams are decoded (at least partially), processed, and, eventually, encoded again. The solutions are complex and decoder dependent. The algorithms allow a large range for the tuning factor *f*, perhaps one of the reasons that guided their implementation. However, this is reached by the dependence of the presentation tool, or by the use of non-real-time computation.

Different from all mentioned work, this paper proposes a time-scaling algorithm for compressed audio streams, simple enough to be executed in presentation time. The algorithm is performed straight on the compressed data, supporting tuning-factor variation, and being independent of the decoder (and thus the player) implementation. Due to the intentional simplicity of the proposed algorithm, its tuning-factor is limited to the range [*0.90* , *1.10*].

Indeed, this paper proposes a framework for a class of algorithms, that is, a meta algorithm. The framework is instantiated for a set of format-dependent algorithms, described in the paper, and implemented as a library, called HyperAudioScaling tool, which can be easily integrated with third-party applications. The media formats handled by the library are MPEG-1 audio (ISO, 1993), MPEG-2 systems (ISO, 2000) and audio (ISO, 1998) (ISO, 1997), MPEG-4 AAC audio (ISO, 2001), and AC-3 (ATSC, 1995). These standards were chosen because they have been largely used in commercial applications, such as those for digital and interactive TV, and also in different audiovisual formats, like VCD, SVCD and DVD.

3 AUDIO TIME-SCALING ALGORITHM

Many high-quality audio formats deal with audio streams as a sequence of frames (or segments). Every *frame* has a *header* and a *data field*, and is associated with a *logical data unit* (*LDU*). A set of coded audio samples, gathered during a small time interval (typically, about 30ms), concatenated with auxiliary bits (called PAD) compose a logical data unit. The number of PAD bits is not limited and are generally used to carry metadata.

Although associated with a specific frame, the LDU does not need to be carried in the data field of this frame. Alternatively, the LDU can borrow bits from data fields of previous frames (the *bit reservoir* in MPEG nomenclature) and be transported partially or entirely in previous frames. The maximum size of the bit reservoir is limited. Thus, data fields can contain one, several or part of an LDU. Figure 1 shows an audio stream with frames separated by vertical lines. In each frame, the header bytes are stripped. They are located in the beginning of the frame and are followed by a data field. The figure also depicts the LDU of each frame.

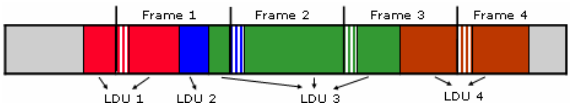

Figure 1: Frame representation of a compressed audio stream.

The HyperAudioScaling algorithm is based on a well-known time-based algorithm called *Granular synthesis* (Gabor, 1946). However, an important difference must be pointed out. Once time scaling must be executed without decoding the compressed audio (to recover the audio samples), and because an LDU can be coded in frequency domain, the chosen adjustment unit is the LDU (and not samples, as in the Gabor's proposal). Thus, HyperAudioScaling

time scaling is performed by removing or duplicating LDUs when the stream must be speed up or slowed down, respectively.

Nevertheless, removing an LDU is not that simple, since the associated frame must also be removed. As a frame may contain LDUs associated with frames that are ahead, these LDUs cannot simply be taken out of the stream, and must be respread into previously scheduled frames.

Similar care must also be taken when duplicating an LDU, since the associated frame must also be duplicated. If a frame has its associated LDU spread into other previous frames, the duplicated frame may also have to distribute part of its LDU. Furthermore, bits of other LDUs within the frame must not be duplicated.

As in both the removing and the duplicating processes there could not be enough room for the LDUs that have to be distributed, HyperAudioScaling tries to avoid using frames where the problem may happen. So, the following rules are adopted:

1. First try to remove only the frames that have their associated LDUs larger than the frame data fields.
2. Similarly, first try to duplicate only frames that have their associated LDUs smaller than their data fields.

The rules ensure that there will always be place to process the selected frame. Note also that new PAD bits may have to be added to previous frames after removal, or to the processed frame in the case of duplication. Figure 2 illustrates the removing process after dropping frame 3 in Figure 1 example. The duplicating process is illustrated in Figure 3 after the duplication of frame 2 in the same Figure 1 example. The figures consider that there are no PAD bits in the LDUs of frames 3 and 2, before removal and duplication, respectively.

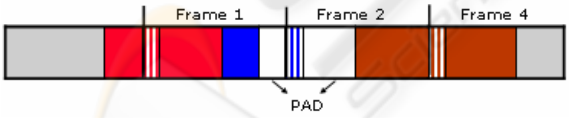

Figure 2: Dropping frame 3 in the stream of Figure 1.

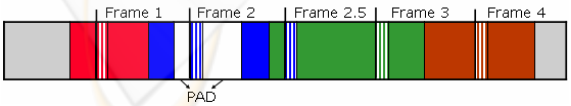

Figure 3: Duplication of frame 2 in the stream of Figure 1.

The added auxiliary bits when rules 1 and 2 are applied can now be used to refine the proposed algorithm. In order to use these PAD bits in future processing, they can be transferred frame by frame, till the next LDU to be processed, using the following algorithm:

- Compute the number *X* of PAD bits which can be transferred from one frame to the next. In this case, the *X* value must be limited to the *bit reservoir* maximum size.
- Transfer *X* bits of the next-frame LDU to the PAD field and insert new *X* PAD bits into the end of the transferred LDU.

Figure 4 shows PAD transference from frame 2 to frame 4 in the stream of Figure 3.

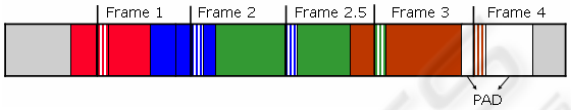

Figure 4: Frames of Figure 3 after PAD transference.

Using this mechanism, new rules can be defined to process frames:

3. If a frame does not satisfy rule *1*, it can still be removed if the length of its associated LDU added to the transferred PAD length is greater than or equal to the frame data field length.
4. If a frame does not satisfy rule *2*, it can still be duplicated if the length of its associated LDU is smaller than or equal to the frame data field length plus the transferred PAD length.

Since there are frames that still cannot be processed, the elastic time adjustments may not be uniformly distributed. However, with the new added rules, more frames can be processed and closer to linear the time scaling can be. A linearity measure of this algorithm will be presented in Section 7. Note also that the non-linearity prevents marked time instants in the original audio to be estimated using a linear equation based on the adjustment factor (see requirement (iv) in Section 1). Therefore, the time-scaling mechanism should track particular time instants and indicates their new values during time-scaling processing.

HyperAudioScaling operation does not require a frame of fixed length. The frame extractor algorithm, instantiated for each audio format, must be responsible for recognizing and creating frames from the stream.

During the audio time scaling, a frame must be properly selected and analyzed to verify if it can be processed. The frame selector processing algorithm is responsible for this task and can be summarized as follows:

a. Compute the effective adjustment factor applied until this moment. This calculus must take into account the number of frames already processed divided by the number of frames analyzed;
b. Compare this result with the original adjustment factor to decide whether the frame should be processed, then verify if the frame can be

processed considering rules 1 to 4, described previously. If it cannot be processed, take the next frame, and return to step (a).

When changes are made in the adjustment factor, the frame selector processing algorithm must recalculate the effective adjustment factor from the moment of the change on.

HyperAudioScaling preserves bytes before the first frame and after the last frame of the stream, because they may represent important metadata, like ID3 standard (Nilsson, 2006) on MP3 streams. Moreover, HyperAudioScaling also maintains metadata inside non-processed frames. Unfortunately, the metadata of the processed frames cannot be always preserved. As some audio formats (MP2, for example) have their LDUs coinciding with their data fields, it is impossible to discard or duplicate frames without their PAD fields. In addition, in other format types, it is difficult to discover where the PAD bits begin without decoding. Fortunately, PAD bits of LDUs usually just carry stuffing bits for length alignment and can be discarded or duplicated without compromising metadata information.

The HyperAudioScaling algorithm was instantiated to MPEG-1 and MPEG-2 BC Audio (MP1, MP2, MP3), AC-3 and MPEG-2/4 AAC. Since there is no *bit reservoir* in MP1, MP2 and AC-3 streams, the instantiations assume that the number of borrowed bytes is always zero, that is, the frame associated LDU is always inside the frame data field. In MP3 streams there is a *bit reservoir* and the algorithm was applied straightly. MP1, MP2 and MP3 have equal-size frames (using constant bit and sampling rate) and auxiliary bytes (PAD) at the end of each LDU, while AC-3 streams have frames with variable length. As previously discussed, this does not affect the algorithm.

The current algorithm implementation for MPEG-2/4 AAC streams uses the ADTS transport protocol. Similarly to MP3, MPEG-2/4 AAC streams provide the *bit reservoir* facility and have frames with both audio and auxiliary data. However, this format introduces some new challenges, since its encoder may use sample values of previous frames to predict sample values of the current frame. Since this encoder facility is optional, it was left to be treated in a future work.

4 TIME SCALING IN SYSTEM FLOWS

Some media formats can multiplex video, audio and metadata in a unique flow called system stream. System streams are usually composed by PACKETs, each one containing one, several or part of an elementary-stream frame.

System time-scale modifications are performed by adjusting each elementary stream and recalculating some metadata within the PACKET header. System time-scaling algorithms usually consist of the following steps:

1. Identify PACKETs from the original stream and demultiplex the input stream into its elementary streams;
2. Convert PACKETs into frames of the corresponding elementary stream.
3. Execute elementary time-scaling algorithms.
4. Recreate system PACKETs using the processed bits and multiplex them creating the new system stream.

The algorithm proposed in Section 3 can be used to adjust elementary audio streams. Another algorithm should be used to process the video stream. However, some changes must be made in the audio time-scaling algorithm. First, time scaling on elementary streams needs to cope with system and elementary stream metadata, since some of the metadata can be affected by the time-scaling processing, such as clock timestamps, etc. Second, time scaling on elementary streams must provide information for adjustment control and inter-media synchronism verification.

Since time-scaling algorithms do not necessarily make linear adjustments, a synchronization mismatch among elementary streams may occur and must be detected by the intermedia synchronism verification function. If the mismatch crosses a specific upper bound[1], the intermedia synchronism verification function must call the services of the adjustment control function to change the tuning-factors of each individual media, in order to correct the problem.

Figure 5 resumes the proposed time scaling for system streams.

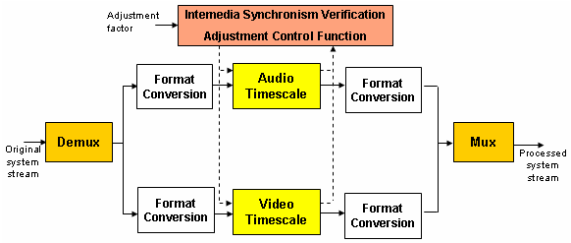

Figure 5: Time scaling for system streams.

[1] Reference (Aly, 2002) discusses the audio and video mismatch problem and states that the loss of quality can only be perceived when adjustments cause a synchronization mismatch higher than 160ms.

The system time-scaling algorithm was instantiated to MPEG-2 system program stream, together with the audio time-scaling algorithm described in Section 3 and the video time-scaling algorithm proposed in (Cavendish, 2005). In this instantiation, the metadata that must be considered for updating are the timestamps PTS (Presentation Time Stamp) and DTS (Decoding Time Stamp), inserted in the elementary streams, and SCR (System Clock Reference), in the system stream. Also, it may be necessary to update the PES_packet_length field in the PACKETs that had their frames processed for elastic time adjustments.

5 AUDIO TIME-SCALING TOOL

The audio time-scaling tool was implemented using the Java language. The time-scaling tool receives requests from client applications (adjustment API). The audio input can be retrieved (pull applications) or received as a stream (push applications). The adjusted stream can be stored in a file or returned to the client application.

Time-scale modifications can be performed at *compile-time* or during *execution-time*. Since these approaches have different characteristics, one API was developed for each one.

Table 1 describes the API to request compile-time adjustments. The *config* method receives the original audio content, a tuning-factor for each specified audio section (set of samples), and the URI of the output file.

The *start* and *stop* methods begin and stop the time-scaling processing, respectively. The *addTimeScalingListener* method allows client applications to register themselves to be notified when the time-scaling processing finishes.

After time scaling finishes, client applications can use the *getTimeScalingInstant* method to discover new time values for marked time intervals; the *getOutputTools* method to get the output file URI; and the *getReport* method to retrieve statistics about time-scaling processing (such as number of frames processed, processing time, etc).

Table 1: Compile-time adjustment API.

Method	Description
config (originalMedia, {tuningFactor, audioSection}, outputFileURI)	Configures the time-scaling tool.
start()	Starts processing.
stop()	Stops processing.
addTimeScalingListener (observer)	Adds a new observer to the time-scaling finish event.
getTimeScalingInstant (timeInstant)	Gets the new time value after adjustments for an original time instant.
getOutputTools()	Gets the output file URI.
getReport()	Gets statistics about the adjustment processing.

Table 2 describes the API for requesting time scaling on the fly. The *config* method receives the original audio, the tuning-factor, and, optionally, a set of time intervals to track. The *setFactor* method can be invoked during time-scaling processing to modify the tuning-factor. If client applications want to be warned when new values of marked time intervals are found, they need to register themselves as observers using *addTimeScalingIntervalListener* method. The *getOutputTools* method must be called by the client application to get the processed stream. The other methods have the same meaning of their compile-time counterparts.

Table 2: Execution-time adjustment API.

Method	Description
config (originalMedia, tuningFactor, {timeInterval})	Configures the time-scaling tool.
setFactor (tuningFactor)	Modifies the tuning-factor.
start()	Starts processing.
stop()	Stops processing.
addTimeScalingInterval Listener (observer)	Adds a new observer to track new values of a marked time interval.
getOutputTools()	Gets the processed stream.
getReport()	Gets statistics about adjustment processing.

6 INTEGRATING HYPERAUDIOSCALING WITH A HYPERMEDIA FORMATTER

In order to illustrate the time-scaling package usage in hypermedia document presentations, the

HyperAudioScaling tool was integrated with the HyperProp formatter (Bachelet, 2004).

When controlling document presentations, the HyperProp may need to change media durations in order to maintain the document temporal consistency. The HyperProp formatter delegates to media players the task of content rendering. To enable incorporation of third-party content players, HyperProp defines an API specifying the methods that media players should implement and how media players should notify presentation events (e.g. user interaction, start/end of a content fragment presentation, etc.). Media players that do not implement the required methods, or do not know how to signalize presentation events, should be plugged to the formatter through adapters.

Based on this approach, the time-scaling package implementation has been used to implement a set of HyperProp time-scaling audio players, as shown in Figure 6. Each audio player is composed by an adapter, the time-scaling tool (Section 5), and a legacy audio player. The adapter receives commands from the HyperProp formatter, executes its tasks and requests the services of the time-scaling tool. The package returns the processed audio stream and dispatches events like the conclusion of time-scaling computation and the discovery of a new time instant. The adapter sends the processed audio stream to the legacy audio players. This legacy player exhibits the processed audio and can dispatch presentation events. These steps can take place both at compile or execution-time.

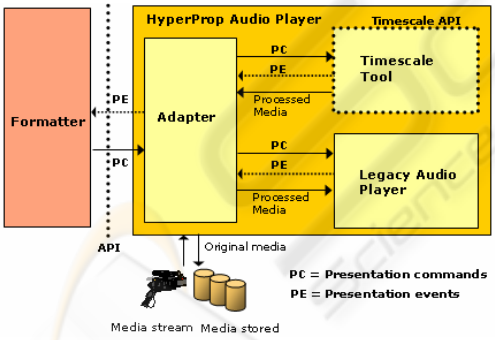

Figure 6: Time-scaling player integrated with the HyperProp formatter.

Two versions of audio presentation tools were implemented using JMF (Java Media Framework) (Sun, 1999). One version works only for compile-time adjustments, using the file generation facility of HyperAudioScaling. The time-scaling tool saves the processed audio in a new file that may be sent to a JMF player after the complete adjustment. To support runtime time-scale modifications another version of audio presentation tool was developed extending the JMF data source. An instance of this new class of data source receives data from the stream returned by HyperAudioScaling and feeds the JMF player for content playing. Since JMF players run as threads of the client application process, it is possible to monitor player events (like user interactions, pause, resume, stop, finish, etc.). One drawback of using JMF is that the set of available codecs are still limited to a few audio formats.

In order to play a wider variety of audio formats, VLC (VideoLan, 2006) player was also integrated with the audio time-scaling tool. Unlike JMF, VLC runs as an external process fired from the client application. As a consequence, it is difficult for the adapter (Figure 6) to interact with VLC players, for example, to monitor events like pause and resume during the audio playing.

7 FINAL REMARKS

This paper discussed time-scaling issues considering specific requirements found in some applications that demand runtime elastic-time short adjustments, in particular the hypermedia presentation systems. From related work analysis, the authors could not find any existing solution that fulfills all the raised requirements.

The proposed time-scaling algorithm was implemented as a library in order to be used by third-party applications. Since, for the purpose of this paper, time-scale modifications are used to manage inter-media synchronization, it normally runs on small stream segments and with a tuning-factor close to 1, what usually brings imperceptible effects to users.

Some subjective and objective simple tests have been performed to measure the algorithm quality and to compare it with the time-scaling algorithm of Sound Forge 8.0. Sound Forge was chosen because of its high quality obtained with the expenses of high processing.

Four tuning factors were used to compare the tools: 0.90, 0.95, 1.05 and 1.10. MP3 44.1 kHz files were used with a compressed audio rate of 128kbps. Ten listeners participated on the test analyzing five audio types.

Subjective notes were assigned by comparing the original file with a processed one. Figure 7 and Figure 8 illustrate the notes obtained. In the pictures, notes are given in MOS (Mean Opinion Score) units (ITU-T, 1998). The audios processed by the HyperAudioScaling algorithm and by the Sound Forge best algorithm are marked, respectively, by

"h" and "s", followed by the used tuning factor (in %).

Although, as expected, the audio quality of the proposed algorithm is worse than the audio quality of Sound Forge, the subjective results showed that they are very close.

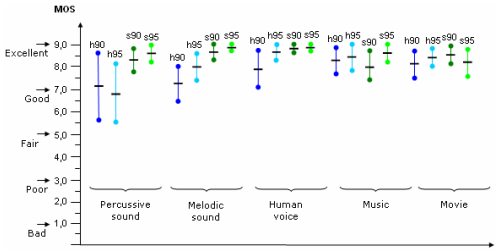

Figure 7: MOS for each media type using 0.90 and 0.95 tuning factors and confidence level of 95%.

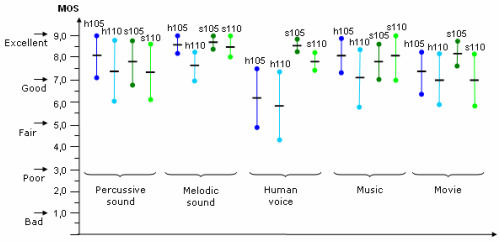

Figure 8: MOS for each media type using 1.05 and 1.10 tuning factors and confidence level of 95%.

On the other hand, the processing rate for MP3 files was about 8-times faster than Sound Forge (2640 frames/sec, using factor 1.1 in a Pentium 4 2.4GHz 1GB-RAM machine). The precision of the proposed algorithm, the reached (actual) audio duration minus the corresponding expected (ideal) audio duration, was always better than, or equal to, the Sound Forge precision for the five different types of audio files. Furthermore, the same tests demonstrated that the audio adjustment algorithm runs almost uniformly along the stream. The worst standard deviation was smaller than half of the distance between two processed frames.

Some media files, processed using HyperAudioScaling algorithm, are available in http://www.telemidia.puc-rio.br/~smbm/ajusteaudio. As future work, we intend to make tests considering different compressed-audio bit rates and also different audio compression standards.

REFERENCES

Aly, S., Youssef, A., 2002. Synchronization-Sensitive Frame Estimation: Video Quality Enhancement. In *Multimedia Tools and Applications*.

ATSC, 1995. Digital Audio Compression Standard (AC-3).

Bachelet, B., Mahey, P., Rodrigues, R.F., Soares, L.F.G., 2004. Elastic Time Computation in QoS-Driven Hypermedia Presentations. *Research Report RR-04-16*, Blaise-Pascal University, Clermont-Ferrand.

Bonada, J. 2002. Audio Time-Scale Modification in the Context of Professional Post-Production. Doctoral Pre-Thesis Work. UPF. Barcelona. Retrieved June 3, 2006, from http://www.iua.upf.edu/mtg/publicacions.php?lng=eng&aul=3&did=219.

Cavendish, S.A., 2005. Ferramenta de Adaptação de Ajuste Elástico em Fluxos MPEG2. *Master Dissertation*, Departamento de Informática – PUC-Rio, Rio de Janeiro, Brazil. (in portuguese)

Covell, M., Slaney, M., Rothstein, A., 2001. FastMPEG: Time-Scale Modification of Bit-Compressed Audio Information. In *Proceedings of the Int. Conference on Acoustics*. IEEE-ICASSP.

Gabor, D., 1946. Theory of Communication. In *Journal of Institution of Electrical Engineers*.

ISO, 1993. Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s - Part 3: Audio, 11172-3.

ISO, 1997. Information technology - Generic coding of moving pictures and associated audio information - Part 7: Advanced Audio Coding (AAC), 13818-7.

ISO, 1998. Information technology - Generic coding of moving pictures associated audio information - Part 3: Audio, 13818-3.

ISO, 2000. Information technology - Generic coding of moving pictures and associated audio information: Systems, 13818-1.

ISO, 2001. Information technology - Coding of audio-visual objects - Part 3: Audio, 14496-3.

ITU-T, 1998. Subjective audiovisual quality assessment methods for multimedia applications, P.911.

Lee, E., Nakra, T.M., Borchers, J., 2004. You're The Conductor: a Realistic Interactive Conducting System for Children. In *Proc. of the NIME 2004 Int. Conference on New Interfaces for Musical Expression*, Japan.

Microsoft. *Windows Media Player*. Retrieved March 20, 2006, from http://www.microsoft.com/windows/windowsmedia/.

Nilsson, M. *ID3v2*. Retrieved March 20, 2006, from http://www.id3.org/.

Omoigui, N., He L., Gupta, A., Grudin, J., Sanacki, E., 1999. Time-Compression: Systems Concerns, Usage, and Benefits. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, USA.

Sony, 2006. *The Sound Forge Product Family*. Retrieved March 20, 2006, from http://www.soundforge.com.

Sun. *Java Media Framework v2.0 API Specification*. Retrieved March 20, 2006, from http://java.sun.com/.

VideoLan Project. *VLC media player*. Retrieved March 20, 2006, from http://www.videolan.org/vlc/.